Technical University of Crete, Greece

School of Electrical and Computer Engineering

Hybrid Visual Simultaneous Localization and Mapping (SLAM) on the Nao Robot using ROS

Nektarios Sfyris

neksfiris at gmail.com

Chania, July 2019

Πολυτεχνειο Κρητης

Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Υβριδικός Οπτικός Εντοπισμός και Χαρτογράφηση (**SLAM**) στο Ρομπότ **Nao** με χρήση του **ROS**

Νεκτάριος Σφυρής
neksfiris at gmail.com

Χανιά, Ιούλιος 2019

Abstract

Simultaneous Localization and Mapping (SLAM) is one of the fundamental problems a robot must solve in order to become truly autonomous. A variety of SLAM methods have been proposed, depending on the available robot sensors for measurements (camera, laser, infrared, lidar, sonar, GPS, compass, etc.) and the available prior knowledge about the environment being mapped and navigated, ranging from controlled environments, such as robotic warehouses, to totally unstructured and unknown terrains, such as the scene of a disaster. In this thesis, we present a Hybrid Visual SLAM approach, implemented and tested on the monocular case of the Nao humanoid robot. The proposed approach combines the benefits of both a Direct (Direct Sparse Odometry or DSO) and an Indirect (Oriented FAST and Rotated BRIEF SLAM or ORB-SLAM) visual odometry method. Specifically, the Direct module provides the total system's initialization process and local camera tracking, while the Indirect module provides relocalization, loop closing and map refinement. In addition, points of the physical three-dimensional space are selected from each module and at each camera keyframe to create the final consistent sparse point cloud map of the environment. All these tasks are executed in a parallel and multi-threaded architecture on a remote computer station, which communicates with the robot over a wired or wireless network. To increase the system's efficiency, we have also included both a geometric and a photometric calibration method to correct the camera measurements. Communication between the Direct and Indirect modules, as well as between the robot and the remote computer station, takes place within the Robot Operating System (ROS) framework, which enables for a common message transmission protocol. Last, but not least, a teleoperation node is built to simulate autonomous robot navigation during SLAM. The coupled system applied to the Nao humanoid robot is evaluated in various indoor and outdoor environments to demonstrate its robustness and real-time performance.

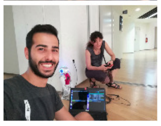
Περίληψη

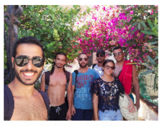
Ο ταυτόχρονος Εντοπισμός και Χαρτογράφηση (Simultaneous Localization and Mapping - SLAM) είναι ένα από τα θεμελιώδη προβλήματα που πρέπει να λύσει ένα ρομπότ για να γίνει πραγματικά αυτόνομο. Ποικίλες μέθοδοι SLAM έχουν προταθεί ανάλογα με τους διαθέσιμους αισθητήρες για μετρήσεις (όπως κάμερα, λέιζερ, υπέρυθρες, lidar, σόναρ, GPS, πυξίδα, κ.λπ.) και ανάλογα με τη διαθέσιμη πρότερη γνώση για το χαρτογραφούμενο περιβάλλον, που κυμαίνεται από ελεγχόμενα περιβάλλοντα, όπως οι ρομποτικές αποθήκες, έως αδόμητες και άγνωστες περιοχές, όπως το σκηνικό μιας φυσικής καταστροφής. Σε αυτή τη διπλωματική εργασία, παρουσιάζουμε μία Υβριδική Οπτική προσέγγιση SLAM, η οποία υλοποιήθηκε και δοκιμάστηκε στην περίπτωση του ανθρωποειδούς ρομπότ Naoπου διαθέτει μία μόνο κάμερα (monocular vision). Η προτεινόμενη προσέγγιση συνδυάζει τα οφέλη τόσο της Άμεσης (Direct Sparse Odometry ή DSO), όσο και της Έμμεσης (Oriented FAST and Rotated BRIEF SLAM ή ORB-SLAM) οπτικής οδομετρίας. Πιο συγκεκριμένα, η Άμεση μέθοδος παρέχει τη διαδικασία αρχικοποίησης του συνολικού συστήματος και την τοπική παρακολούθηση της θέσης της κάμερας, ενώ η Έμμεση μέθοδος παρέχει επαναπροσανατολισμό της κάμερας, κλείσιμο βρόχου και βελτίωση του χάρτη. Επιπλέον, επιλέγονται σημεία του φυσικού τρισδιάστατου χώρου από κάθε μέθοδο και για κάθε εικόνα του βίντεο της κάμερας, για να δημιουργηθεί ο τελικός χάρτης νέφους σημείων του περιβάλλοντος. Όλες αυτές οι διεργασίες εκτελούνται σε μία παράλληλη και πολυνηματική αρχιτεκτονική σε απομακρυσμένο υπολογιστή, ο οποίος επικοινωνεί με το ρομπότ μέσω ενός ενσύρματου ή ασύρματου δικτύου. Για να αυξήσουμε την αποδοτικότητα του συστήματος, έχουμε συμπεριλάβει και μία μέθοδο γεωμετρικής και φωτομετρικής βαθμονόμησης της κάμερας για τη διόρθωση των μετρήσεών της. Η επικοινωνία μεταξύ της Άμεσης και Έμμεσης μεθόδου, καθώς και μεταξύ του ρομπότ και του απομακρυσμένου υπολογιστή, πραγματοποιείται μέσα από το Robot Operating System (ROS), το οποίο επιτρέπει ένα κοινό πρωτόκολλο μετάδοσης μηνυμάτων. Τελευταίο στοιχείο, αλλά εξίσου σημαντικό, είναι η προσθήκη ενός κόμβου τηλεχειρισμού για να προσομοιώνει την αυτόνομη πλοήγηση του ρομπότ κατά τη διάρκεια της διαδικασίας SLAM. Το πλήρες σύστημα εφαρμοσμένο στο ανθρωποειδές ρομπότ Nao αξιολογείται σε διάφορα εσωτερικά και εξωτερικά περιβάλλοντα για να επιδειχθεί η σταθερότητα των αποτελεσμάτων και η αποδοτικότητά του σε πραγματικό χρόνο.

Acknowledgements

First, I would like to thank my advisor Michail G. Lagoudakis for his trust and guidance during the course of this thesis.

Next, the members of team “Kouretes”, namely Dimitris X., Tasos K., Michalis A., Fotis L., Thanasis B., Dimitris E., Athanasia K., George K., George A., Hercules T., Maria K., Panagiotis G., Errikos S., Olympia G., Spiros P. and Sotiris K. which i hope will continue contributing to the team’s goals and will further develop what we have to this day built together. Special thanks to Helen Tsagkarogianni for her support and assistance to this thesis experiments conduction.





My friends Charis S., Maria D., Giannis M., with special thanks to Alex T. and Giannis P. for their support and contribution to the experiments completion.

Last, but not least, I would like to thank my family for their love and constant encouragement. Without their great efforts i wouldn’t be able to complete this thesis.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The first and most fundamental problem that needs to be solved by a robot in order to become really autonomous, is the problem of Simultaneous Localization and Mapping (SLAM). This problem has attracted a lot of attention and a lot of progress has been made the last two decades by many Universities, technological companies and research centers, as the modern CPUs-GPUs, with the cooperation of quality sensors and optimized algorithms allow to solve it in real-time. The idea of SLAM, can find application to many emerging technologies, such as autonomous aerial robot rescue operations, ground robot explorations, and underwater aquaculture research. Various ways exist to approach SLAM, such as using LiDAR laser scanning systems, an array of cameras, or even a set of radar sensors.

We use the term Visual SLAM when the sensor used to find the robot's location and motion in the map it builds, is a camera sensor setup, which is usually built as a stereo vision system consisting of two cameras with known field of view, and distance between them. The family of methods that enable a robot to estimate the SLAM problem using vision, is called Visual Odometry (VO). Traditionally in VO, features, also called landmarks, are extracted from the observed scene, and their correspondence is being searched in subsequent image frames to explain what the robot's motion and the geometry of what it sees are. This group of methods are referred to as Indirect VO. On the other hand, a more straightforward group of methods exists, that try to achieve the same goal but by only examining the brightness values of the pixels in each image. This second group of methods are referred to as Direct VO. When a single camera is used as the only sensor to understand the Structure from Motion (SfM) in a sequence of images, we have the fully-constrained case of Monocular Visual SLAM.

The goal of Visual SLAM, is to estimate the robot's camera trajectory, while at the same time reconstruct the observed environment in a map. Because depth is not observable by a single camera, and can only be approximated up to a scale, drifts occur that affect the robustness of the whole procedure. A technique that is used today in many systems, despite how computationally costly it may be, to compensate for this issue, is called Bundle Adjustment (BA). If enough map points match, and consecutive camera poses are provided, it can refine a whole area of estimations helping the robot to relocalize itself in the map in case it is lost, or find a place that it has already visited, and thus successfully complete a loop closure in the map.

1.1 Thesis Contribution

In this thesis we describe an implementation of a combined direct and indirect Visual Odometry system, based on the system proposed by Hun Lee and Civera in [1], which aims to solve the problem of Simultaneous Localization and Mapping problem on the Nao robot model using a single camera. For each of the two subsystems, the best choices in their respective group of methods up to date were selected based on performance; namely, the Direct Sparse Odometry (DSO) system of Engel, Cremers, and Koltun [2] for the direct one, and the ORB-SLAM system of Mur-Artal and Tardos [3] for the indirect one.

These two subsystems create and maintain their separate maps in real-time, which are then coupled in a way to work complementary to each other and produce a more accurate semi-dense Point Cloud map. In particular, DSO is used to estimate the robot's camera pose in a local area robustly, while also providing its initialization method for the whole system, and points from its own map when the need to dense up the coupled map exists. In parallel, ORB-SLAM tries to make refinements to bigger scale camera trajectories, search and close loops, and relocalize the robot in the map it creates. All these tasks are executed on a highly-parallel and multi-threaded architecture on a remote computer.

We include both a geometric and a photometric calibration scheme for our monocular camera, to enable for robust camera tracking, and accurate points depth estimation, both achieved by correcting online the projected sensor measurements on the image plane.

The communication between the two subsystems and with the Nao robot takes place within the Robot Operating System (ROS) framework [4], that allows for a common message transmission protocol and real-time operation of the coupled system. In compatibility with this framework, a teleoperation node is built to simulate an autonomous robot navigation that aims to explore an unknown environment and create a map so it can be later used for complex tasks. Last but not least, the overall system is tested in mostly planar ground environments, in order to measure its efficiency and performance.

1.2 Thesis Outline

In Chapter 2 we present sufficient background information needed for this thesis. We give an overview of the Aldebaran Nao humanoid robot, the Robot Operating System (ROS), and the general ideas behind localization, mapping, and their coupled problem SLAM. We also provide basic information on how the camera measurements are processed, and about the Bundle Adjustment method. In Chapter 3 we state the problem of Visual SLAM and Visual Odometry and we refer to different approaches that have been used over the last two decades. In Chapter 4 we describe in detail our proposed approach; specifically, the network specifications used to conduct our experiments, the methods for calibrating our camera both geometrically and photometrically, and the coupled system's functionality. In Chapter 5 we present results of the implemented system on different types of environments and on various scenarios. Lastly, Chapter 6 acts as an epilogue for this thesis, presenting our conclusions about the work done, along with future improvements.

Chapter 2

Background

2.1 Aldebaran Nao Humanoid Robot

Aldebaran Nao is an integrated, programmable humanoid robot, designed by a French robotics company, Aldebaran Robotics, and is now further developed and manufactured by Softbank Robotics[1]. Nao robot's development began in 2004, and in 2007 Nao officially replaced Sony's Aibo quadruped robot in the RoboCup Standard Platform League[2] shown in 2.1[3], an international robotics competition in which research teams program their autonomous Nao robots to compete in soccer matches. The robots must also be able to locate themselves in the soccer field of predefined dimensions, thus solving the problem of *Localization*. This is a subproblem of the Simultaneous Localization and Mapping (SLAM) we want to achieve for our Nao robot in this thesis.

On a technical view, Nao V5 consists of an ATOM Z530 processor at 1.6 GHz clock, 1 GB of RAM, and 2 GB of Flash memory running an Embedded GNU/Linux distribution based on Gentoo, giving it the ability to execute simple computatiolally on-board tasks. It is powered by a 6-cell Lithium-Ion battery, able to store 48.6 WattHours in total, providing 60 to 90 minutes of continuous operation, and has an IEEE 802.11g network card, which makes it able to communicate wirelessly or with a RJ45 wired Ethernet link.

Being a robot of 58 cm in height, 28 cm in width, and weighing 5.4kg, it is a robot with various sensors and actuators. In more detail, it has 26 motors of 4 different types

[1]https://www.softbankrobotics.com/
[2]https://spl.robocup.org/
[3]https://www.flickr.com/photos/robocup2013/

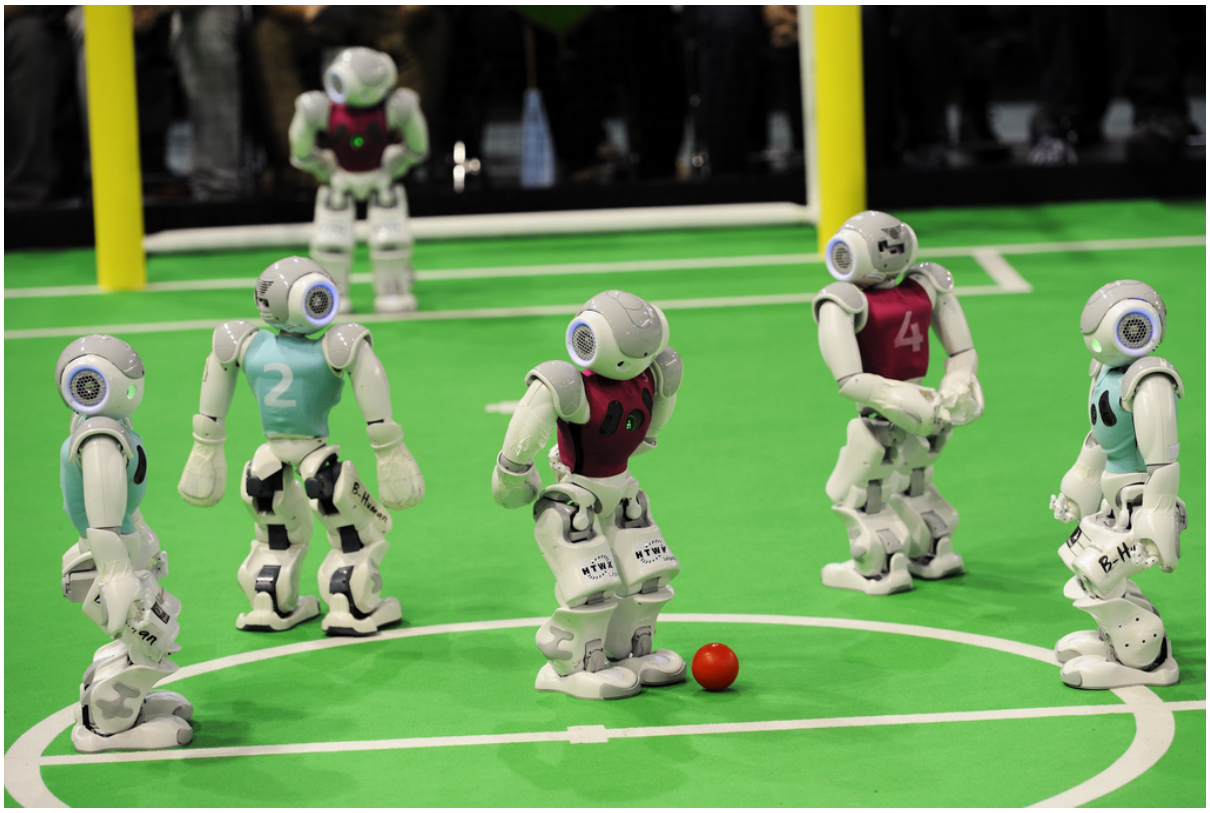
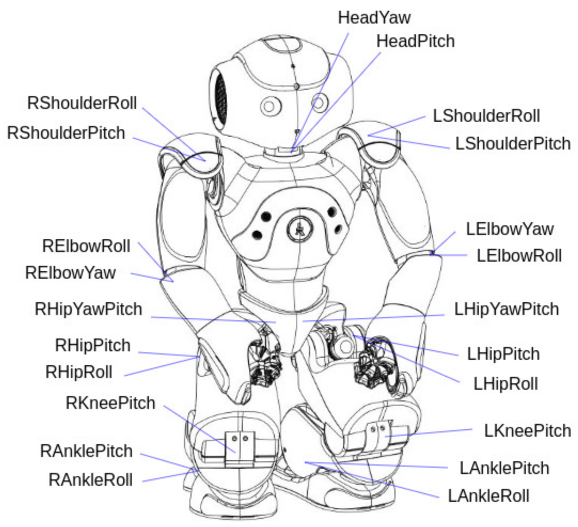


Figure 2.1: Standard Platform League

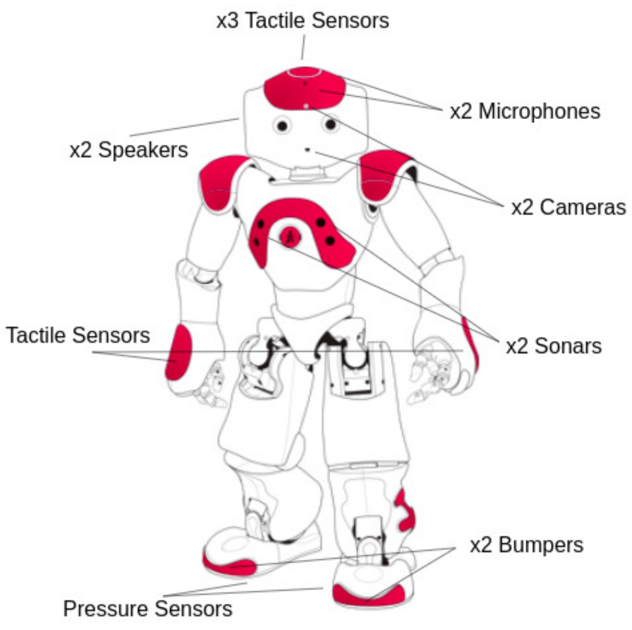
in its body (with differences in rpm, stall torque and nominal torque), providing it with 25 degrees of freedom; 2 in the head, 4 in each arm and 2 in every hand, 5 in each leg and 1 in the hip joint for yaw, which is enabled by the inclined and coupled rotary axis joint. Each joint is equipped with a Magnetic Rotary Encoder for position feedback, providing 12 bit precision, corresponding to 0.1^o precision. Figure 2.2a shows the position of the motors.

Likewise, Figure 2.2b shows the sensors' position Nao robot is eqquiped. Two identical cameras providing non-overlapping views and up to 1280×960 resolution at 30fps, two ultrasonic sensors with an effective cone of 60^o and a detection range around 0.5 m, an inertial unit consisted of a 3-axis gyrometer and a 3-axis accelerometer, 4 microphones, touch sensors, force sensitive resistors and tactile sensors.

Currently, Nao robot has been through six version upgrades. The operating sys-



(a) Nao Motors' position



(b) Nao Sensors' position

Figure 2.2: Aldebaran Nao Robot

tem of the robot is an embedded GNU/Linux distribution that coexists with NAOq[1], the main software thats runs on the robot, controls it, and therefore gives life to it. The NAOqi framework allows homogeneity in programming, communication of different modules and information sharing. It is cross-platform, cross-language and provides introspection, meaning that the framework can monitor which functions are available and where. It also supports parallelism, resources, synchronization, and events.

NAOqi OS supports software development in C++, Python, Java and Javascript after the appropriate SDK installations. Finally, Webots[2] is an open-source robotic simulator used in several online robot programming contests, including RoboCup Standard Platform League, and therefore supports Nao robot model for testing behaviors on existing or custom environments.

[1] http://doc.aldebaran.com/2-5/index_dev_guide.html
[2] http://doc.aldebaran.com/2-1/software/webots/webots_index.html

2.2 Robot Operation System (ROS)

The Robot Operation System (ROS)[1] [4] started in 2007 as an outgrowth of the STanford Artificial Intelligence Robot and the Personal Robotics Program from Stanford University, and continued developing under Willow Garage[2]. ROS is an open-source BSD-licensed middleware, created to simplify the development, design, and maintenance of multi-sensory systems' applications, and generalize the communication protocol used. It provides not only standard operating system services, such as hardware abstraction, package and process management, but also high-level functionalities, such as asynchronous and synchronous calls, and centralised database.
Overall, the main concepts of ROS are:

- The ROS **Master** is the first service to run so that the whole ROS ecosystem starts operating. It is implemented via XMLRPC, a stateless and HTTP-based protocol that is relatively lightweight, and does not require a stateful connection. The Master also provides naming and registration services to nodes, while tracks them to topics and services, giving them the ability to find each other and communicate.

- The **Parameter Server** is part of the Master and is also implemented in the form of XMLRPC. It is a shared database between nodes to recover and store static, non-binary data at runtime. In brief, the Parameter Server can store basic XML-RPC scalars, lists, base64-encoded binary data, and dictionaries.

- **Nodes** are the main computational units and execute different operations (mapping, path planning, etc.). A robotic system usually contains several nodes, which communicate with one another peer-to-peer, by streaming topics or calling services. There are two types of nodes, the *Subscriber*, when a node receives information from a topic, and the *Publisher*, when the node broadcasts information to a topic. Of course, there can be nodes who are both Subscriber and Publisher.

- **Topics** are declared buses with a unique name that transport information. One or more nodes can either publish data to a topic, or one or more nodes can subscribe to one. In fact, nodes can't know who published data or subscribed to a topic. The

[1] https://www.ros.org/
[2] http://www.willowgarage.com/

type of information transported by a topic is a message, and each topic is strongly connected to it. It is also important to mention that topics are an asynchronous way for nodes to stream data.

- **Messages** are simple or complex data structures containing type fields. For ROS messages, standard data types (such as integer, double, etc.), are supported. For instance, a message can contain image or any other sensor data. At last, there is used an interface definition language (IDL) to describe the messages published through topics.

- **Services**, unlike Topics, exchange data synchronously and represent a small procedure that a node shall do. They support Remote Procedure Call (RPC) request/reply interactions, and are defined by a pair of messages, one for the caller of the service, and one for the responder. In general, a node advertises a service, offering it a name, so another node can locate it and make a request. Namely, a service can be used for obtaining the current battery level.

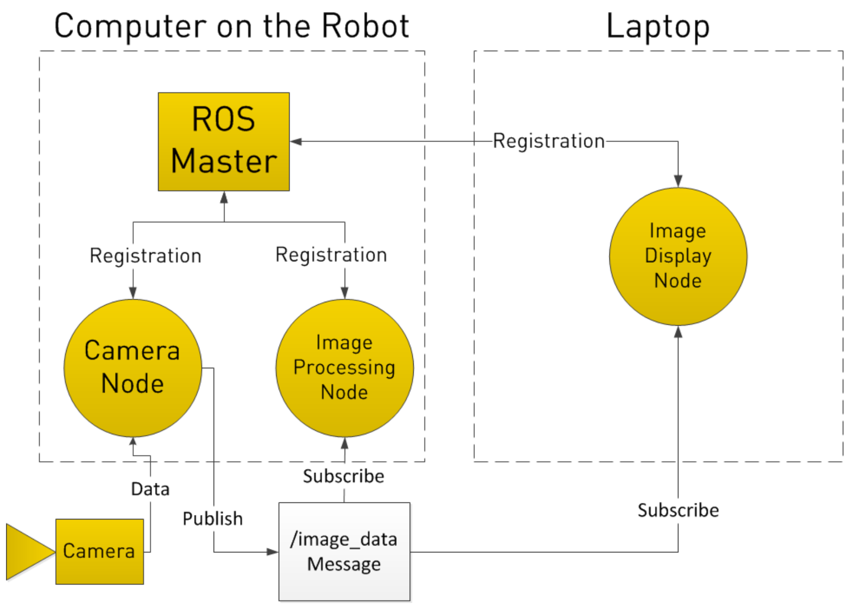


Figure 2.3: A high level view of a ROS system example

A simple example, similar to how we used ROS in our work, is shown in Figure 2.3[1]. The ROS environment begins with the ROS Master running on the computer of the robot. One Master is only allowed in ROS, in contrast with the new updates that ROS2 version will bring. The Master allows the nodes to register and locate each other. Here we have two nodes on the robot and one more on a different computer system, a laptop. Among them, a *Camera Node* that communicates with the camera by subscribing to the topic the camera publishes (we will name it /image_raw topic), and then publishes those data to a topic called /image_data. A *Image Processing Node* and a *Image Display Node*, subscribing to the topic /image_data published by the Camera Node. *Image Processing Node* wants the camera data to do some image related computations, and *Image Display Node* to display the raw images on a screen. The overall communication between the robot and the foreign computer system can happen under a TCP/IP protocol or even an Ethernet protocol.

We can see what the message of /image_raw and /image_data topics consists of in the current example, by looking at the Table 2.1.

Because our two topics publish a stream of data, each instance of the data must have a way to be identified and differentiated from every other set. That's why a Header in the message is needed, containing the unique number of sequence, timestamp, and frame_id, that each set of data belongs to. Additionally, the message contains the image height, which is the number of rows, and the image width, the number of columns. The encoding of the pixels, meaning the channel's type, ordering and size, if the image is bigendian, the image step, which gives you the distance in bytes between the first element of one row and the first element of it's next row, and at last the actual data of the image in a matrix, sizing $step \times rows$.

Strengths of ROS are also that it supports a peer-to-peer architecture, enabling each node to communicate directly with any other in a synchronous or asynchronous way and can record or playback the messages. It is language-neutral, meaning that it can be developed with many programming languages, such as C++, Python and Lisp. It is thin and easy to control, since libraries, drivers, and algorithms are kept as executables, making them reusable while ROS's size is contained as small as it can be during execution. The

[1] https://robohub.org/ros-101-intro-to-the-robot-operating-system/

code written for it is reusable, and can be applied with other robot software frameworks. And of course, it is tool based, with its microkernel design allowing for using tools to build, run, and manipulate the various ROS components, and is suitable for large robotic or generally multi-sensory systems. It is notable, that for different Linux distribution, corresponds different ROS distribution.

To summarize, we used the ROS framework in our work because of its usability, strengths compatibility with our system's development, open-source philosophy, and wide range of tools, like package debugging and bug tracking, robot visualization with Rviz, and coordinates transformation manipulation system tf. In addition, ROS provides us with an already fully developed driver for Nao Robot 2.1, *naoqi-driver* [1], which is basically a NAOqi OS module that bridges with ROS and translates the NAOqi messages to ROS messages. The module is written in C++, and offers low latency in message transportation and real-time CPU usage, by collecting several sensor data of Nao robot straight from the lowest levels of NAOqi.

A sensor_msgs/Image ROS Message

std_msgs/Header *header*
 uint32 *seq*
 time *stamp*
 string *frame_id*
uint32 *height*
uint32 *width*
string *encoding*
uint8 *is_bigendian*
uint32 *step*
uint8[ ] *data*

Table 2.1: A ROS message definition of an uncompressed image.

2.3 Localization and Mapping

2.3.1 Mobile Robot Localization

Robot localization is the problem of determining the position of a robot with respect to a map of its environment. It is one of the fundamental issues that a robot must solve during an autonomous navigation, as a good estimate of its own, or a wanted object's, location and pose, can allow for quality decisions, leading to good future actions. The

[1]http://ros-naoqi.github.io/naoqi_driver/index.html

information needed for a robot to localize itself and establish a relation of its local coordinate frame to the global coordinate frame, comes from its own perception and motion sensors. Perception sensors, provide information from observations of the environment and therefore can be used to understand the differences in it (e.g camera images), while motion sensors provide data related to the robot displacement and overall motion drift (e.g odometry data).

There are three categories of localization problems, as described in the Probabilistic Robotics book of Thrun, Burgard, and Fox in [5], which are identified by the type of information that is available during navigation at the initial state, and during the runtime:

- **Pose Tracking**. In Position tracking, we assume that an initial estimate of the robot pose is known, and by updating this estimate with the last sensor measurements, we aim to localize its current position. As time passes, the robot pose error accummulates and position tracking becomes more difficult. That's why both sensors that provide robot pose information in relation to itself, and in relation to the global map are needed. On the whole, the goal is to find the correspondence between the sensor measurements and the model of the environment, in order to compensate incremental errors, and maintain a reliable belief about the robot pose. Position tracking is a local problem, as the uncertainty is calculated based to the area near the robot'As true pose.

- **Global Localization**. Tracking can be used when an initial position is given, but in real life robotic scenarios, it is usually not the case. The problem when the robot has little, or no information about its initial pose, and has to place itself in the map, is known as global localization or pose initialization. Because of the absence of boundaries in uncertainty, and the multimodal approaches that help with the estimation, global localization is more difficult than pose tracking, and actually subsumes it.

- **Kidnapped robot problem**. This problem is a variant of the global localization problem and refers to the relocation or teleportation of an operating robot away from its previous position inside the map. Without any notification, it must be able to recognize such a change and adapt to it by correctly updating its pose. It is a more difficult task than global localization, since the robot has the belief that it knows where it is, while it isn't there. The practical importance of the kidnapped robot problem, is that is gives the chance for testing new localization algorithms' ability to recover from global localization failures, as an even more general problem.

The type of the environment can also have a substantial impact on the difficulty of the localization:

- *Static environments.* In Static environments, we assume that there is no change in the environment but the pose of the robot. These environments have simple mathematical properties, and thus are convenient for probabilistic estimation.
- *Dynamic environments.* Here, other objects, including the robot, are moving inside the map. This has a long-term influence on the robot's perception. Usually, changes that are not easily measurable, affect only a single measurement, or have big variance in their values, are treated as noise. As obvious, dynamic environments are closer to real life scenarios.

After the data of the sensors are collected, and because of the existing high uncertainty in motion and perception, probabilistic models are usually used, formulating the localization problem as a Bayasian estimation problem.

Robot Pose

To be able to control the movements and actions of a robot, we need to have a good understanding of its pose, relative to itself or another coordinate system. The pose, or kinematic state, is defined by the orientation and position of the robot in the global coordinate frame. Rigid robots moving in a 3D map, are usually described with six state variables (or seven, depending on the method for expressing orientation). On the other hand, if we deal with a more simple version of that problem, meaning rigid mobile robots that move in a 2D planar environment, the robot's position is represented with a matrix containing its **x** and **y** Cartesian coordinate pair, and its angle of orientation, or bearing, with θ. At a random time t, we have:

$$\mathbb{R}^2 : \mathbf{X}_t = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{2.1}$$

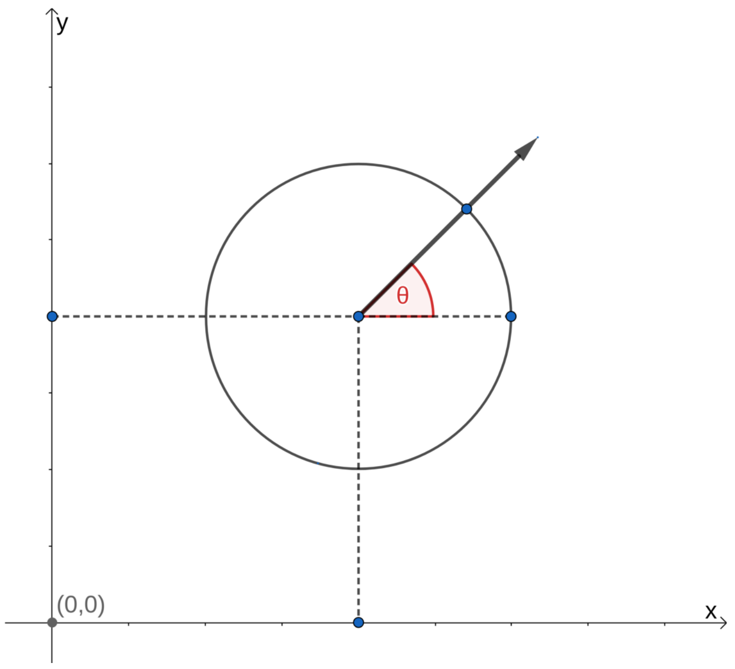


Figure 2.4: Robot pose in two dimensions

A graphical example of a robot pose in a planar environment can be seen in Figure 2.4. Also, considering the noise that is included in a robot's pose estimation, we can represent the belief it has about its pose, by a probability distribution that expresses the uncertainty related to it, denoted as $bel(\mathbf{x}_t)$.

Motion Model

Mobile robots must have actuators (hydraulic, pneumatic, electric, or mechanical) in order to move in their environment. The ability of a robot to move is deeply connected to its localization solving problem, as it contains important information about its current pose. In practise, we find two types of motion models differentiated by the sensor information, providing the robot pose, available, the *Odometry-based*, and the *Velocity-based* motion model. Odometry-based models are used when a robotic system is equipped with actuator encoders, and therefore can get the needed information about each actuator's pose through them, while Velocity-based models are applied when there are no encoders available, so they calculate the new pose based on the time elapsed of the different velocities provided. In general, odometry models tend to be more accurate than velocity models.

Continuing, even if robots have a direct control over their actuators, their influence to their actual locomotion is more complex, as actuators or sensors can suggest error because of heat, or even hardware malfunction. During time, the locomotion action information that took place is available at the end of each time step by issuing with the robot controls, and given the current state of the robot and the current control input, the robot can make a transition to a new state. Summarizing, we can describe the motion model of a robot with a probabilistic formula as,

$$P(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) \tag{2.2}$$

Here, $\mathbf{x}$ represents the robot pose, while $\mathbf{u}$ a motion command. This model is describing the posterior distribution of the robot pose $\mathbf{x}_t$ in time t, when the motion command $\mathbf{u}_t$ that is to be executed, and the robot pose at an earlier time $t-1$ are known. Note that this motion model adopts the memoryless and stochastic process of the *Markov property*, as each transition depends only on the last robot robot state and current action.

Sensor Model

Of what we've seen until now, in order to solve the localization problem of a robot moving inside a map, a motion model is needed. But the motion model alone can't give us a good estimate of the robot's real pose, and it gets even more difficult when we are talking about the robot's global localization or the kidnapped robot problem. Therefore, external sensors are needed that can provide with additional information for a more precise estimation. These sensors can be cameras, range sensors, or even touch sensors, and due to sensor uncertainty caused by errors, a probabilistic formula is appropriate for modelling them,

$$P(\mathbf{z}_t|\mathbf{x}_t, \mathbf{m}) \tag{2.3}$$

Here, x is the robot pose, $\mathbf{z}$ is the perceptual information or measurement at time t, and $\mathbf{m}$ is the map of the robot's environment. This sensor model describes the likelihood of making the observation $\mathbf{z}_t$ at time t, while the pose $\mathbf{x}_t$ at the same time, and the map that includes the robot are given. Like with the motion model, this model also adopts the Markov property. In practice, it is often impossible to model a sensor accurately, as the development of that model can be really time-consuming, and because the model can include state variables that have been omitted by mistake or are unknown, such as the exposure of an image.

2.3.2 Robotic Mapping

To use the collected measurements from a robot's sensors, it is needed to specify the environment, or the map, inside which the measurements were generated. We have described the localization problem, as the robot's effort to find its pose, meaning its location and orientation, in an already given map. There exist real-world cases that providing the map, inside which the robot will move, is legitimate, such as a warehouse specially built for robots to store and retrieve objects. However, there are scenarios that providing the map of the environment can not be possible, either because of the lack of a portion of information, or the complete unawareness of its nature and obstacles, such as the ruins of a building.

The ability of a robot to create a map of its environment provides it with real autonomy, as it can adapt to changes of it, and doesn't need human supervision. The main challenges that a robot will face during the mapping problem are:

- The **Hypothesis space**. It consists of the set of all possible approximations, or maps, that a robot can create. Because of the continuous growth in size, and therefore space, of the maps during navigation, the hypothesis space gets remarkably larger in dimension. Three main concerns for choosing a hypothesis space are, its *size*, meaning the number of options-hypotheses available to choose from, *randomness*, if it is stochastic or deterministic, and the type, number, relationship between the *variables-parameters*.

- **Learning maps**. When neither a map exists nor the pose is known, the robot has to simultaneous make efforts to understand its position and orientation in space, as well as map the world around it, based on the belief it has about its pose. Because, a bad estimation in pose, can lead to false positioning of an object in the map, and therefore a bad map. The localization and mapping (also known as

SLAM, described in more detail in the Section 2.3.3 depend on each other, for an autonomous robot navigation.

- **Noise**. While the robot navigates in its environment, it accumulates errors in odometry and other perception sensors, making the overall problem more difficult. The difficulty further increases, when the size of the environment that the robot has to map, gets larger too.
- **Perceptual ambiguity**. A mobile robot can come over different places that look alike, which can lead to mistakenly establishing wrong correspondences between different locations.
- **Loop closing** is an important skill a robot must have in the mapping process, as if it can notice an already crossed location in the map, it will be able to correct past mistakes, in ambiguity, or accumulated sensor model errors.

We can express the map, as the collection of the objects, linked with their respective location, inside it,

$$\mathbf{m} = \{m_1, m_2, ..., m_N\} \tag{2.4}$$

Here, each object $\in 1 \leq n \leq N$, and each $m \in m_1 \leq m_n \leq m_N$, are each object's specific attributes. Usually, we are dealling with two types of maps, either *feature-based*, or *location-based* maps.

Occupancy Grid Mapping

Occupancy grid maps belong to the family of *location-based* maps, and this family of maps, considers the n of the Equation (2.4) to be a specific location.

Most occupancy grid maps used in practice, are 2D floor plan maps, representing a slice of the 3D world. In their 2D grid, each cell, or pixel, posseses an occupancy value, specifying the probability that the area which the robot is mapping, is occupied by an object. This value of likelihood of occupation ranges from 0 to 1, with 0 corresponding to no object occupation, while 1 to a certain occupation of an object. Figure 2.5 describes the Occupancy grid map model we discussed, but with the addition of an intermediate value, $0,5$, for areas that the robot is unsure about their occupation. Here, the value 0 is coloured with white, 1 with black, and $0,5$ with grey. Usually, the more dark a cell is, it represents that it is more likely to be occupied.

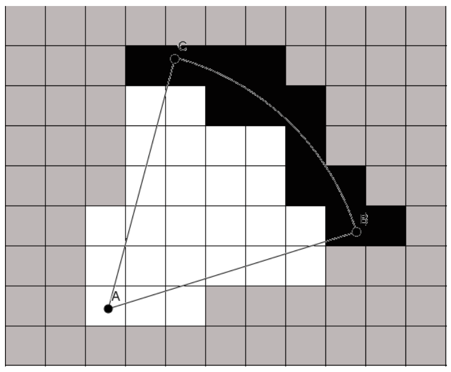


Figure 2.5: Occupancy Grid Mapping Model

The goal of any occupancy grid mapping algorithm is to compute the posterior over a map, depending on the already given knowledge about the robot state,

$$P(m|\mathbf{z}_{1:t}, \mathbf{x}_{1:t}) \tag{2.5}$$

As we have already seen, m is the map, $z_{1:t}$ is the set of all of the robot's sensor measurements up to time t, and $x_{1:t}$ is the set of all its poses until t, also called as the *path of the robot*. As the occupancy grid map splits space in finite i grid cells, we can describe it with,

$$\mathbf{m} = \sum_i m_i \tag{2.6}$$

Here, $\mathbf{m}_i$ denotes the grid cell with index i of the map m, and is assigned with an occupational probability value $p(\mathbf{mi})$ value in the range $[0,1]$. The occupancy grid philosophy can be generalized to three dimensions, but the computational complexity is very high, meaning that it can slow down real-time mapping, and therefore can only be applied under permissible conditions. For example, if we allow the occupational probability of each grid to strictly take a binary value, either 1 or 0, then the number of different maps that can be represented by a map of 1000 cells (which is a small number for a map), equals 2^{1000}. This means that it is not possible to compute a posterior probability for every possible map. Therefore, by breaking down the general problem, we try to estimate the posterior of each grid cell created in the map and expand Equation (2.5) above, as,

$$P(\mathbf{m}_i|\mathbf{z}_{1:t}, \mathbf{x}_{1:t}) \tag{2.7}$$

In this way, we get rid of the the high dimensional posterior of Equation (2.5), but create a new problem, as we can not be represent the possible dependencies of neighboring

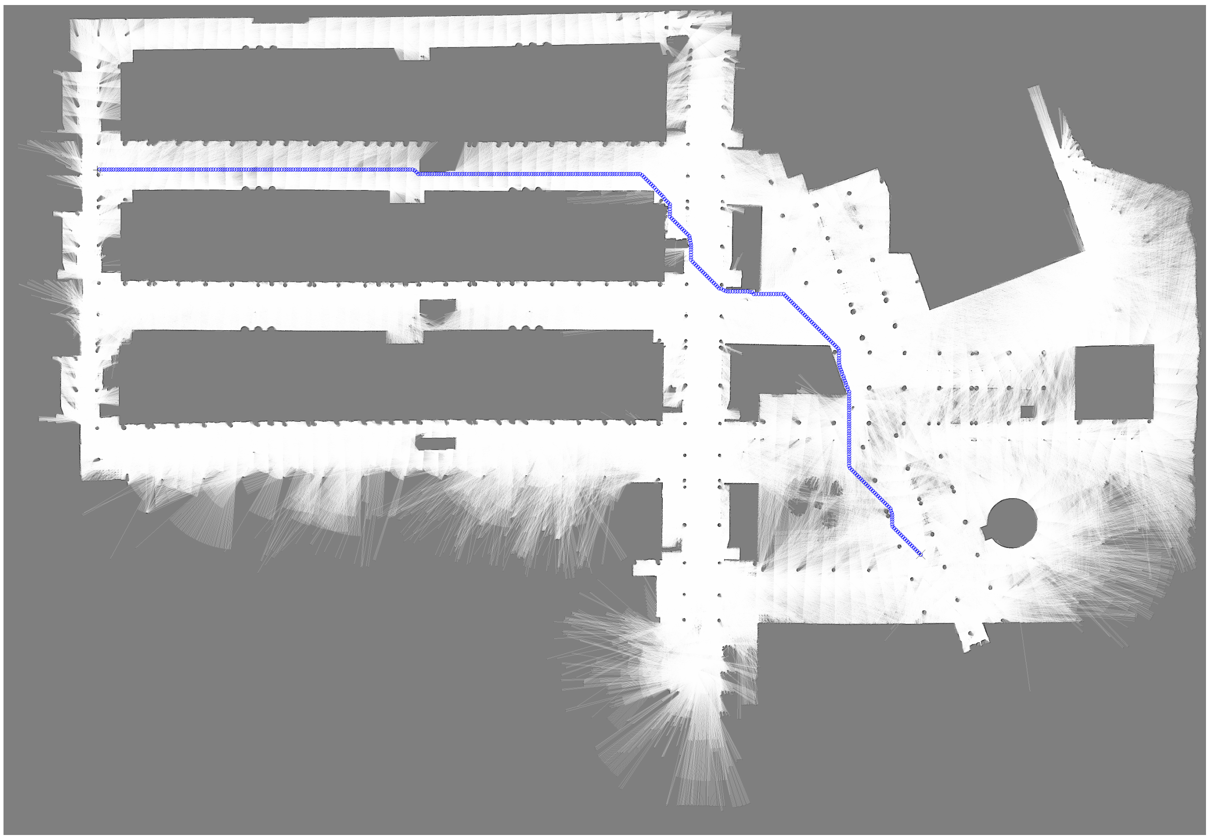


Figure 2.6: Occupancy grid mapping on a large environment

cells. So, the posterior of a map is estimated as the product of the probabilities of all cells inside of it,

$$P(m|z_{1:t}, x_{1:t}) = \prod_{i=1} P(m_i|z_{1:t}, x_{1:t}) \tag{2.8}$$

Concluding, as shown in Figure 2.7[1], occupancy grid maps have a graphically simplistic way of representing the robot's environment, making it easy to find paths through the unoccupied space, and thus can suggest effective path planning algorithms.

[1]https://www.mrpt.org/tutorials/programming/path-motion-planning/path_planning_over_occupancy_grid_map/

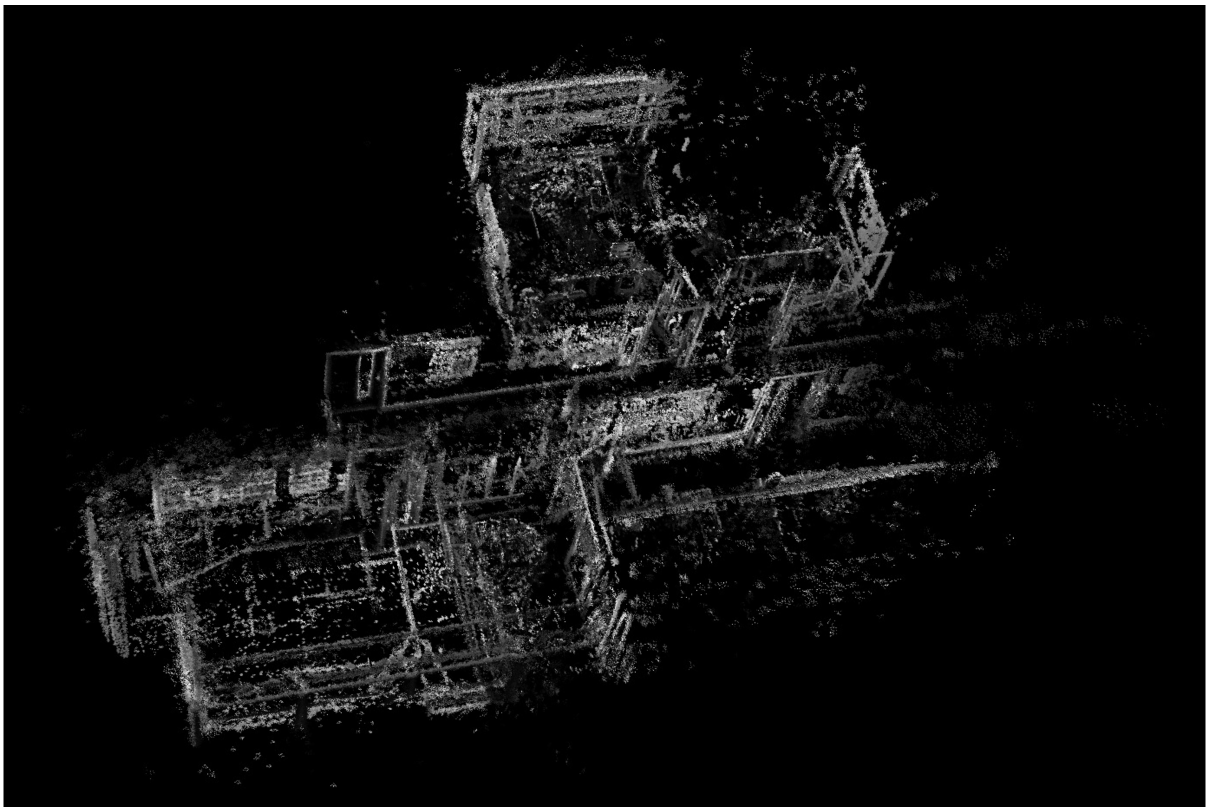


Figure 2.7: Point Cloud mapping on a large environment

Point Cloud Mapping

In the real world, a *feature* corresponds to a distinct object, or a specific structure in the environment. Examples of often used features are edges, patterns on objects, and textures on walls. In robotics, features are also called *landmarks*, as they can represent unique locations in the map, usually to indicate their usage for robot navigation. Landmark-based models are usually defined in the family of *feature-based* maps, which consider n in the Equation (2.4) a feature index, and the value of m_n contains the properties of a feature, and the Cartesian location of the feature. During mapping, a robot can use the variety of features it has extracted to build a map, by positioning each individual feature in a 3D, empty map at first, as a point. These maps are called *Point Cloud Maps*, as they consist of multiple points in their space to represent the robot's environment. Figure 2 shows a point cloud map as created from the TUM Mono VO Dataset [6] by running the DSO system [2] we describe in Chapter 4.

Features are usually obtained by range or vision sensors with a *feature extractor*, and usually, there is a further need to measure the distance between their position, and the robot's local coordinate frame. If we assume the feature extractor to be a function f, then the features extracted are given by $f(\mathbf{z}_t)$. Also, note that a *feature vector* is an n-dimensional vector, conveniently describing numerical features as,

$$f(z_t) = \{f_t^1, f_t^2, ...\} = \left\{ \begin{pmatrix} r_t^1 \\ \phi_t^1 \\ s_t^1 \end{pmatrix}, \begin{pmatrix} r_t^2 \\ \phi_t^2 \\ s_t^2 \end{pmatrix}, ... \right\} \tag{2.9}$$

Here, r is the range to the feature relatively to the robot's position, ϕ is the bearing, or angle of orientation, and s is the feature's signature, which can be for example its average color.

The main advantage of feature extraction, is the enormous reduction of computational complexity, as an initial set of variables in the high dimensional space, is reduced to a low dimensional feature space, while still accurately describing the original data set. In doing so, the dimensionality reduced of the sensor measurements, can reach several orders of magnitude.

In general, point cloud maps can only describe the shape of the environment, in a more sparse, or dense way. That means, we view objects as a collection of multiple points in the 3D space, which can not always provide us with enough information about the object's identity, but makes these types of maps very efficient for real-time mapping. Also, as they depend greatly on sensor data, they can provide multiple sensor estimations about the location of a specific object, or landmark, and therefore can locate it much easier in space, making the method a good choice for robot navigation.

2.3.3 Simultaneous Localization and Mapping

The simultaneous localization and mapping, namely SLAM, problem, assumes that the robot is not given a map of its environment, and has no knowledge about its poses beforehand. Therefore, it must do the estimation of its pose and the environment's map at the same time. SLAM is a fundamental problem for robots to become truly autonomous, and inside the robotics community it is considered very complex and hard, as a map is needed for localization, and a good pose estimate is needed for mapping.

From a probabilistic perspective, there are two main categories of SLAM problems:

- The *full SLAM*, seeks of calculating the posterior over the entire path $x_{1:t}$ and the map of the mobile robot,

$$P(x_{1:t}, m | z_{1:t}, u_{1:t}) \tag{2.10}$$

- The *online SLAM* insted, deals with the estimation of the posterior over its current pose x_t, and the map m at time t,

$$P(x_t, m | z_{1:t}, u_{1:t}) \tag{2.11}$$

  Like with the $fullSLAM$, the measurements $z_{1:t}$, and the controls $u_{1:t}$ are known. Additionally, for more efficient computations, many online SLAM algorithms discard past sensor measurements and controls once they have been used, to boost real-time processing. In relation with the full SLAM, the online SLAM is the result of integrating out past poses, meaning that the Equation (2.11) can be written as,

$$\int\int ... \int p(x_{1:t}, m | z_{1:t}, u_{1:t}) dx_1 dx_2 ... dx_{t-1} \tag{2.12}$$

  These integrations are usually computed one at a time during the SLAM.

Another important attribute of SLAM, comes from its nature, as it contains a *continuous* and a *discrete* component. The continuous estimation gives priority to the location of the objects, or landmarks represented in a feature form, and the robot's own pose variables, inside the map. On the other hand, the discrete estimation prioritises the correspondences and relations of objects in the map. In this case, when an object is observed, the SLAM algorithm will keep operating by searching its identity in relation with other already detected objects.

In total, we could say that the idea behind *online* and $full$ SLAM, is calculating the posterior over the map and the robot's path. In reality though, estimating the posterior seems infeasible, as the continuous parameters we work with live in a high-dimensional space, and the number of the discrete correspondence variables is pretty large. There are also cases that the feature correspondences are unknown, leading to an exponential growth of the correspondence vector $u_{1:t}$, storing all possible assignments of correspondences.

Visual SLAM

Visual SLAM is a quickly developing technology, as a subcategory of the general SLAM problem, in the embedded and computer vision community. It refers to the process of determining the position and orientation of a sensor, often represented as a rigid-body (discussed in Section 2.4.1), or a local coordinate system of a robot, with respect to its surroundings, by using visual sensors, such as a monocular, a stereo camera, or a RGB-D camera.

Some of the reasons visual SLAM has seen so much progress in recent years are, the vast information a camera can provide in little time, its low size, weight, and power (SWaP) footprint, and that it is cheap. You can have low-cost and agile robots, able to solve difficult problems in robotics and machine vision.

On the technical side, most visual SLAM systems work by tracking map points through successive camera frames, aiming to triangulate their 3D position, while simultaneously using this information to approximate the camera pose. It is therefore, a joint estimation of camera motion and 3D location, also called *structure and motion*.

In general, visual SLAM systems are aiming to minimize a reprojection error, or the difference between the projected points and their actual position, by often using *Bundle Adjustment* (further analyzed in Section 2.6).

2.4 Camera Measurement Model

In this Section we will provide fundamental information on how the measurements received from a camera, in our case a monocular setup, are being processed in order to understand the camera's motion in 3D space. A lot of what we will see from now on in this Chapter is influenced by D. Cremer's lectures [7]. In three-dimensional reconstruction, we aim to create a 3D representation of the world, from a set of 2D projections of the environment. The geometric relations between the 3D scene and its 2D projections, are based on two main types of transformations:

- The *Euclidean*, or *Rigid body motion*, which accounts for the motion of a 3D coordinate system from one frame to another.

- *Perspective projection*, the procedure responsible for representing three dimensional objects on a picture plane.

2.4.1 Rigid Body Motion

When we talk about a *Rigid* object, we mean the consistency in distance between any two internal points on a complex body, as it moves around. For example, a robot can be represented as a rigid body, as all of its joints (arms, legs etc.) will keep belonging to its body and move together as the robot moves in space. A rigid body's movements belong in the 3D space, and we will try to represent them by extending the theory behind the 2D pose, described in Section 2.3.1.

We can say, that every point that lives in the three dimensional Euclidean space $\mathbb{E}^3$, is characterized by the following coordinates in time t, so that $\mathbb{E}^3$ can be identified with $\mathbb{R}^3$,

$$\mathbf{X}_t = \big(x, y, z\big)^\top \in \mathbb{R}^3 \tag{2.13}$$

By not considering time t, we can also represent a *bound vector*, pointing from a point X to a point Y at any time,

$$v = \mathbf{Y} - \mathbf{X}, \quad v \in \mathbb{R}^3 \tag{2.14}$$

Inside the $\mathbb{R}^3$, we can define a product that can map two vectors to another vector, the *cross product*,

$$\times : \mathbb{R}^3 \times \mathbb{R}^3 : \quad u \times v = \begin{pmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{pmatrix} \in \mathbb{R}^3 \tag{2.15}$$

The outcome of the above cross product is an orthogonal vector to u and v, and as it is not symmetric, it can describe an *orientation*. If u is fixed, then a linear mapping between the two vectors is introduced, as $v \mapsto u \times v$, and can be described as a skew-symmetric matrix,

$$\hat{u} = \begin{pmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{pmatrix} \in \mathbb{R}^{3\times3} \tag{2.16}$$

In the same way, every skew symmetric matrix $\in \mathbb{R}^{3\times3}$ can be identified with a vector $u \in \mathbb{R}^3$. The hat ($\wedge$) operator specifies an *isomorphism* between the space $\mathbb{R}^3$ and the space of all skew symmetric matrices $so(3)$. The relation of these spaces by using the inverse of the $\wedge$ operator is, $\vee : so(3) \to \mathbb{R}^3$.

Considering all that, we will start creating an idea about what a *rigid body motion* is. On a high level view, it is a family of maps, so that,

$$g_t : \mathbb{R}^3 \to \mathbb{R}^3, \quad \mathbf{X} \mapsto g_t(\mathbf{X}) \tag{2.17}$$

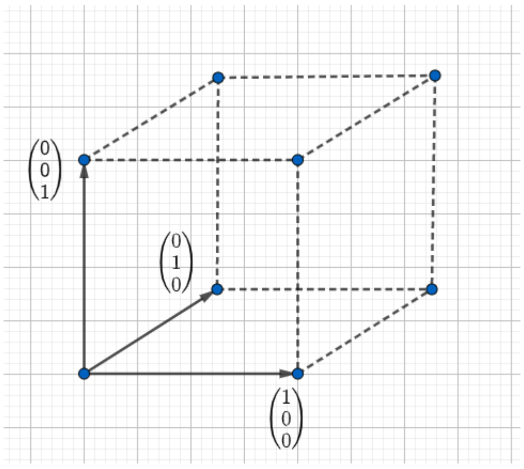
and are able to preserve the norm, providing the length, and the cross product, so angles are preserved, $\forall$ vector $u, v \in \mathbb{R}^3$, as,

- $\|g_t(v)\| = \|v\|$
- $g_t(u) \times g_t(v) = g_t(u \times v)$

$$g_t(x) = Rx + T \tag{2.18}$$

Rigid body motions also preserve the inner product and the triple product, which can be described as $\langle u, v \times w \rangle, \quad \forall u, v, w \in \mathbb{R}^3$, since norm and scalar product can be represented with the *polarization identity*[1], and therefore can preserve *volume* (an example of volume is seen in Figure 2.8). A motion in the 3D space that belongs to a rigid body, is a motion that consists of a translation T and an orientation R of the moving body, and can be described as,

Figure 2.8: A block with volume 1


$\begin{pmatrix}0\\0\\1\end{pmatrix}$
$\begin{pmatrix}0\\1\\0\end{pmatrix}$
$\begin{pmatrix}1\\0\\0\end{pmatrix}$

The motion of the rigid body g_t, since it preserves lenth and orientation, can be well described by a Cartesian coordinate frame assigned to an object, like a camera, given by

[1]https://en.wikipedia.org/wiki/Polarization_identity

the orthonormal oriented vectors $e_1, e_2, e_3 \in \mathbb{R}^3$ and the origin. Thus, the motion, based on the origin, can be described with a translation $T \in \mathbb{R}^3$, where the transformations needed for the orthonormal vectors e_i, are given by the new basis vectors, which also preserve the scalar and cross product,

$$r_i = g_t(e_i) \tag{2.19}$$

The matrix $R = \big(r_1, r_2, r_3\big)$ belongs to the group of $SO(3) = \{R \in \mathbb{R}^{3\times3} \mid R^\top R = RR^\top = I,\ \det(R) = +1\}$, because it is an orthogonal matrix, and its orientation is preserved while not including mirroring ($\det(R) = -1$). For a brief example, if a rotation is $R = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$, and we want to find the appropriate values that describe a wanted rotation, these values of the matrix R must respect the properties of the $SO(3)$ group as we saw right above, as we can't have any given values. With such a rotational matrix, we can transform points in 3D space from their original position (we usually use $R(0) = I$) to another, as,

$$\mathbf{X}_{trans}(t) = R(t)\mathbf{X}_{orig} \tag{2.20}$$

Continuing, if we want to create a differential equation providing us with the approximation of a rotation R, we need a skew symmetric matrix $\hat{w}$, with $\hat{w}(0) \in so(3)$, so that,

$$R(dt) = I + \hat{w}(0)dt \tag{2.21}$$

Considering two skew symmetric matrices $\hat{w}, \hat{v}$, we can further define the *Lie bracket*, or *Lie product* as,

$$so(3) \times so(3) \to so(3), \quad [\hat{w}, \hat{v}] = \hat{w}\hat{v} - \hat{v}\hat{w} \tag{2.22}$$

Lie Group and Lie Algebra

Because a lot of problems in robotics and computer vision are engaged with manipulation of 3D geometry, we need a robust and efficient framework to represent and work with 3D transformations. A tranformation has properties such as invertion, diorientation, or

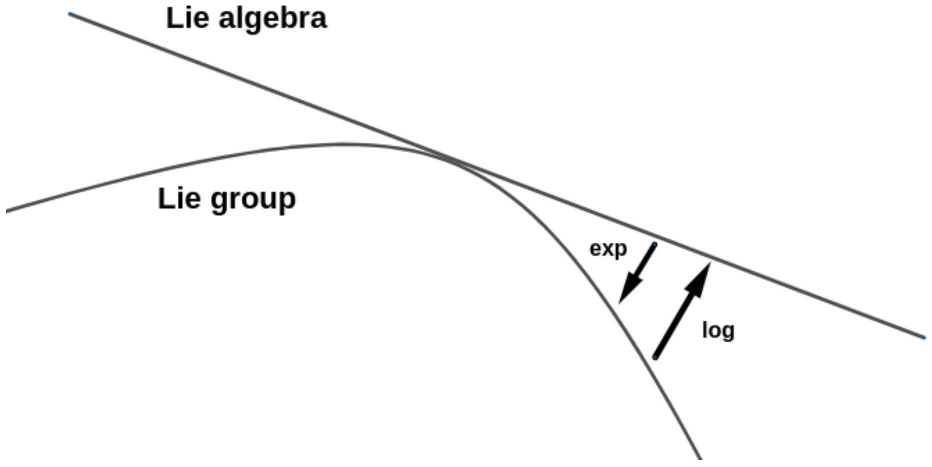


Figure 2.9: Lie Group and Lie Algebra visualization

interpolation, and *Lie groups* can satisfy those operations. In essence, from what we have seen, we can approximate a rotation $R \in SO(3)$, with an element that belongs in the skew symmetric matrices space, $\hat{w} \in so(3)$. There are two main concepts that we need to clarify:

- The **Lie group**, which is a topological group that is also a smooth manifold, that can allow for group operations, multiplication and inversion, to be smooth maps. The rotation group $SO(3)$ is a Lie group.
- The **Lie algebra**. Associated with every Lie group is a Lie algebra, which is the tangent space at the identity I of the lie group. The basis elements of a tangent space are called *generators*, and all vectors on the tangent space describe linear combinations of these generators. The Lie algebra of the Lie group $SO(3)$, is $so(3)$.

The creator of Lie algebra was Marius Sophus Lie[1], a mathematician born in Norway. Importantly, the relation between a *Lie group* and its associated *Lie algebra* is also graphically shown in Figure 2.9, and allows for calculations in one to be mapped successfully to the other.

[1]https://www-history.mcs.st-andrews.ac.uk/Biographies/Lie.html

Transforming from Lie Algebra to the Lie Group

In case we have a known skew symmetric matrix $\hat{w}$, and we want to find an appropriate representation of a rotation $R(t)$, we first have to describe the differential equation system,

$$\begin{cases} \dot{R}(t) = \hat{w}R(t) \\ R(0) = I \end{cases} \tag{2.23}$$

with $\dot{R}$ telling us how R changes if we move away from the identity I on the $SO(3)$. Then, to find the solution, which is the wanted rotation R, we use the **exponential map**,

$$R(t) = e^{\hat{w}t} = \sum_{n=0}^{\infty} \frac{(\hat{w}t)^n}{n!} = I + \hat{w}t + \frac{(\hat{w}t)^2}{2!} + \dots \tag{2.24}$$

From the exponential map $e^{\hat{w}t}$, $w \in \mathbb{R}^3$ is the axis of rotation, and scalar t is the angle of rotation. We can formulate it as,

$$exp : so(3) \to SO(3), \quad \hat{w} \mapsto e^{\hat{w}} \tag{2.25}$$

Transforming from Lie Group to the Lie Algebra

The inverse function of the exponential, as we know from mathematics, is the **logarithm**. Thus, if we know the rotation matrix $R \in SO(3)$, and we want to find a skew symmetric matrix $w \in \mathbb{R}^3$, we will find it from:

$$\mid w \mid = \cos^{-1}\left(\frac{trace(R) - 1}{2}\right), \quad \frac{w}{\mid w \mid} = \frac{1}{2\sin(\mid w \mid)} \begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix} \tag{2.26}$$

Meaning that any orthogonal transformation R, can be represented by an angle of rotation $\mid w \mid$, and an axis of rotation $\frac{w}{|w|}$. There is no unique $\hat{w}$ to model the rotation, but there is a whole set, or family, of equivalent rotations.

Representation of Rigid Body Motion

Like we described in the high-level Equation (2.18), a rigid body motion consists of a translation T and a rotation R, and we can define its space with the help of the special

Euclidean transformations group,

$$SE(3) \equiv \{g = (R, T) \mid R \in SO(3), \quad T \in \mathbb{R}^3\} \tag{2.27}$$

and in *homogenous coordinates*,

$$SE(3) \equiv \left\{ g = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix} \mid R \in SO(3), \quad T \in \mathbb{R}^3 \right\} \subset \mathbb{R}^{4\times4} \tag{2.28}$$

To have a better view about the spaces presented, in the space of points $\mathbb{E}^3$, we are able to see points transform and rotate, and in the space of vectors $\mathbb{R}^3$, we can see only rotations. If we consider a family of rigid body transformations $g : \mathbb{R} \to SE(3)$, we can define the Lie algebra $se(3)$ of **twists** $\hat{\xi}$ elements, also called *exponential coordinates* for $SE(3)$, as,

$$\dot{g}(t)g^{-1}(t) = \hat{\xi}(t), \quad \hat{\xi} \in \mathbb{R}^{4\times4} \tag{2.29}$$

The translated Lie algebra $se(3)$, serving as the tangent space on the identity of the Lie group $SE(3)$, is computed by the derivative, and is the first order aproximation,

$$se(3) \equiv \left\{ \hat{\xi} = \begin{pmatrix} \hat{w} & v \\ 0 & 0 \end{pmatrix} \mid \hat{w} \in so(3), \quad v \in \mathbb{R}^3 \right\} \subset \mathbb{R}^{4\times4} \tag{2.30}$$

Here, $\hat{w}$ is a skew symmetric matrix and w as the *angular velocity* provides rotation, and the *linear velocity* v is a 3D vector providing translation. And we can move between a *twist* and its *twist coordinates* with the hat $\wedge$ and its inverse $\vee$ operators as,

$$\hat{\xi} \equiv \begin{pmatrix} v \\ w \end{pmatrix}^{\wedge} \equiv \begin{pmatrix} \hat{w} & v \\ 0 & 0 \end{pmatrix} \in \mathbb{R}^{4\times4}, \quad \begin{pmatrix} \hat{w} & v \\ 0 & 0 \end{pmatrix}^{\vee} = \begin{pmatrix} v \\ w \end{pmatrix} = \xi \in \mathbb{R}^6 \tag{2.31}$$

The outcome of the second equation lives in the 6-dimensional space and provides us with 3 translation parameters, and 3 rotation parameters. As with $so(3)$, we can use the exponential map to map from the Lie algebra $se(3)$ to the Lie group $SE(3)$, $\hat{\xi} \mapsto e^{\hat{\xi}}$, as,

$$g(t) = e^{\hat{\xi}t} = \sum_{n=0}^{\infty} \frac{(\hat{\xi}t)^n}{n!} \tag{2.32}$$

There is no unique $\hat{\xi} \in se(3)$ to model the rigid body motion $g \in SE(3)$, but there is a whole set, or family, of equivalent twists. We use the first part of the Equation (2.31) when we want to do camera pose estimation, which is equivalent to a rigid body, and therefore need to find its translation and orientation. We first get the $\hat{\xi}$ and then use the *exponential map* to give us the rigid body motion and the wanted camera movement.

Accordingly, we are now able to represent a body's motion, as a robot's specific coordinate frame, for example its camera. From the camera's initial position X_0 in the world frame, considering its motion, we find the new position at time t with,

$$X(t) = R(t)X_0 + T(t) \tag{2.33}$$

and while knowing g from Equation (2.28), we can describe between two different time frames t_1, t_2, the camera motion in homogenous coordinates as,

$$X(t_2) = g(t_2, t_1)X(t_1) \tag{2.34}$$

Here, $X(t_1)$ are the camera coordinates at t_1, and $g(t_2, t_1)$, is the rigid body motion of camera from time t_1 to t_2. We must note that in Equation (2.33), X_0 denotes a 3-dimensional vector, while in Equation (2.34), $X(t)$ is in homogenous coordinates, described as $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$.

Move from Frame to Frame

There is often the necessity in Lie groups to transform from a tangent vector of a tangent space, to another tangent space. In case a camera changes its pose in the map relative to another camera, we can represent that relation as,

$$\mathbf{Y} = g_{xy}X(t) \tag{2.35}$$

with Y being the coordinates in another camera, g_{xy} the rigid body transformation between the camera poses, and $X(t)$ the camera whose coordinates are known. The **adjoint map** helps us perform the transformation we want, and because of the Lie groups' property, this transformation is linear. The relative points detected by a camera, are described by a *twist*,

$$\hat{V}_y = g_{xy}\hat{V}{g_{xy}}^{-1} \equiv \mathbf{ad}_{g_{xy}}(\hat{V}) \tag{2.36}$$

Summarizing, the *adjoint map* allows us to transfer from a frame to another when moddeling velocities, and we will describe it as follows,

$$ad_g : se(3) \to se(3), \quad \hat{\xi} \mapsto g\hat{\xi}g^{-1} \tag{2.37}$$

Rotations Representation

A more intuitive method to represent the rotations of a rigid body is with the **Euler angles**, which are the three angles giving the three rotation matrices. A mathematical approach to define them is by assuming some basis $(\hat{w_x}, \hat{w_y}, \hat{w_z})$ of the Lie algebra $so(3)$, to help us with the rotations on each of the 3 axis x (roll), y (pitch), z (yaw),

$$w_x = \begin{pmatrix}1\\0\\0\end{pmatrix}, \quad w_y = \begin{pmatrix}0\\1\\0\end{pmatrix}, \qquad w_z = \begin{pmatrix}0\\0\\1\end{pmatrix} \tag{2.38}$$

Then, we can map to the Lie group $SO(3)$ as,

$$a : \big(a_1, a_2, a_3\big) \mapsto exp(a_1\hat{w_x} + a_2\hat{w_y} + a_3\hat{w_z}) \tag{2.39}$$

The coordinates a_1, a_2, a_3 are called *Euler angles.* Also, if we say that the angle of rotation around $x - axis$ is ψ radians, around $y - axis$ is ϑ radians, and around $z - axis$ is ϕ radians, the three rotation matrices can be given by:

$$\begin{aligned} R_x(\psi) &= \begin{pmatrix}1 & 0 & 0\\ 0 & \cos\psi & -\sin\psi\\ 0 & \sin\psi & \cos\psi\end{pmatrix} \qquad R_y(\vartheta) = \begin{pmatrix}\cos\vartheta & 0 & \sin\vartheta\\ 0 & 1 & 0\\ -\sin\vartheta & 0 & \cos\vartheta\end{pmatrix} \\ R_z(\phi) &= \begin{pmatrix}\cos\phi & -\sin\phi & 0\\ \sin\phi & \cos\phi & 0\\ 0 & 0 & 1\end{pmatrix} \end{aligned} \tag{2.40}$$

2.4.2 Monocular Camera Measurement Model

We can trace the origins of perspective projection, back in 400 B.C from Euclid of Alexandria. A field of study, that has led to today's *projective geometry.* In order to have a complete view about how a camera sees and understands the physical world, we will begin by describing the most basic camera model, the **pinhole camera model**.

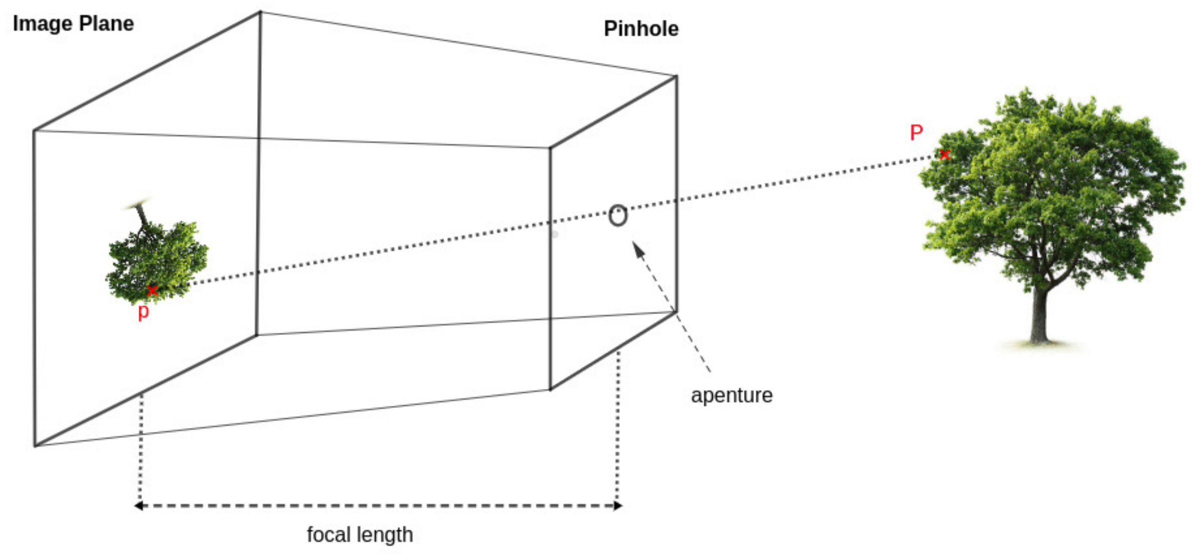


Figure 2.10: The Pinhole Camera Model

The *pinhole camera* is basically a simple camera without lens but with a tiny apenture, called *pinhole*. The idea is that the light rays that are correctly aligned from a scene pass through the pinhole, and are then projected as an inverse image on the image plane, also known as the *camera obscura*, and a natural optical phenomenon. An example of a pinhole camera is presented in Figure 2.10.

In an effort to strengthen the incoming light, and not only accept the aligned light rays of a 3-dimensional point P in the world coordinate frame, with coordinates $X = \big(X, Y, Z\big) \in \mathbb{R}^3$ to the projection plane, we will extend the *pinhole camera model* to the *thin lens camera model*. As the name implies, this model has the addition of thin lens. Here, the perspective projection π from a point P relative to the reference frame centered at the optical center, and with optical axis of the lens being the axis that passes through the optical center, is obtained by comparing the similar triangles A and B:

$$\frac{Y}{Z} = -\frac{y}{f} \Leftrightarrow y = -f\frac{Y}{Z} \tag{2.41}$$

The minus "-" in this equation (expressed in metric units) is justified because a point P in the scene, in comparison to the point p that is projected to the image plane is anti-diametrical. We can expect more rays from point P to be gathered, meaning more

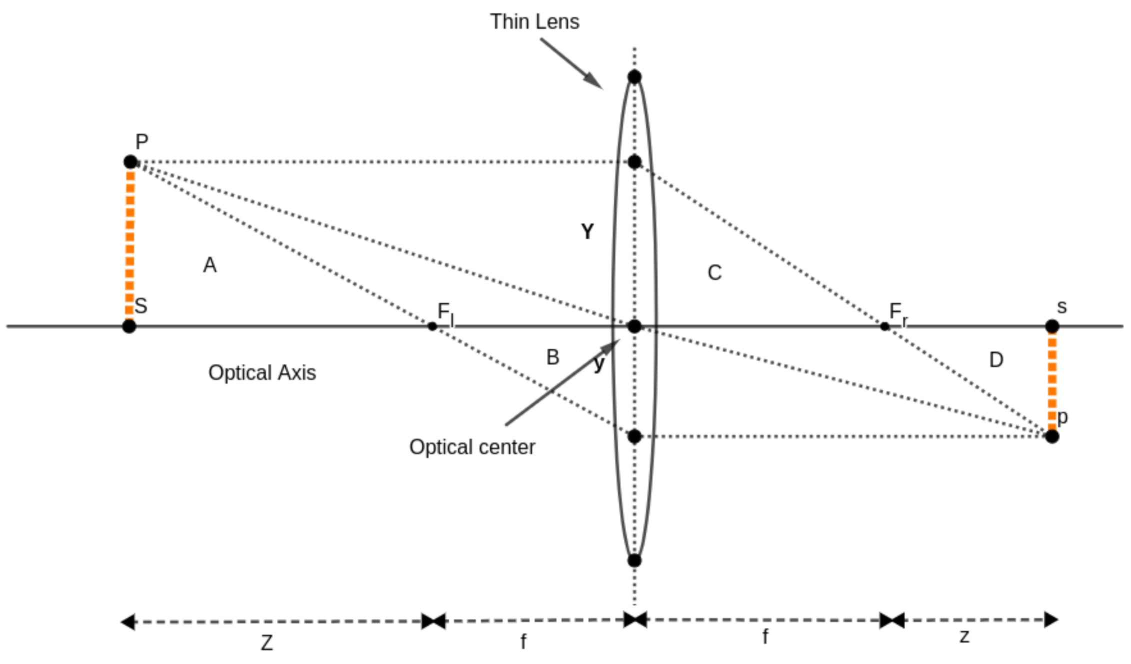


Figure 2.11: Thin Lens Camera Model

information, to create its projected point p. The model described above, is presented with more details in Figure 2.11.

To properly continue with the consept of *perspective projection*, we will make a quick explanation on the main terminology of the field:

- The **optical center**, is the point of the camera where all rays pass in to the image plane.
- The **image plane**, is the plane where the image of the environment is projected, and therefore formed.
- The **principal axis**, is the perpendicular line passing from the optical center to the image plane.
- The **principal plane**, is the plane on which the optical center is located, and is parallel to the image plane.

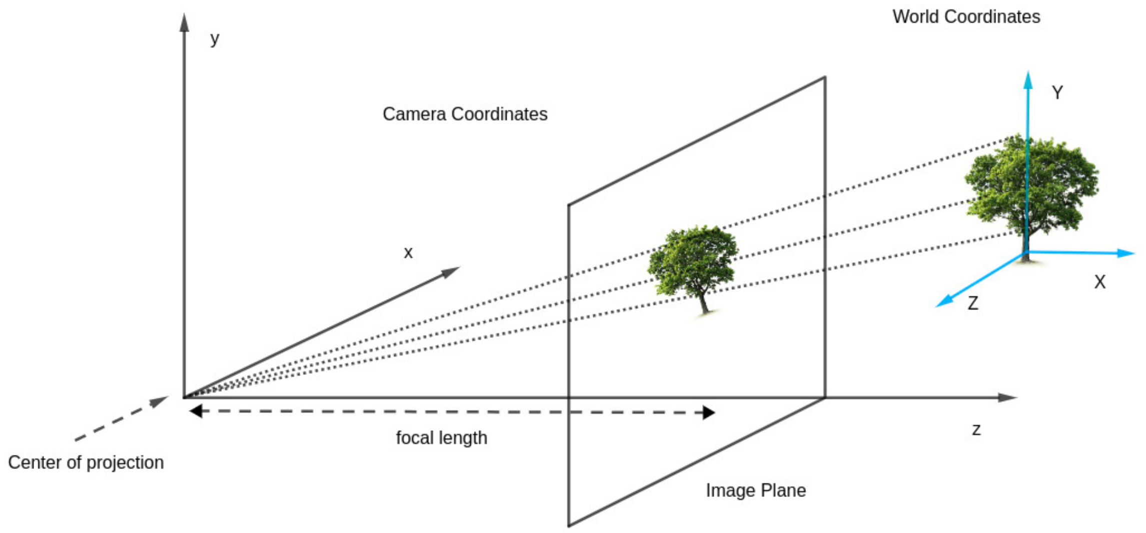


Figure 2.12: Rearranged Pinhole Camera Model

- The **focal length** f, is the distance between the optical center and the image plane. In case of the pinhole camera model, it is the distance from the camera apenture to the image plane.

To simplify the mathematical part, we can consider the image plane to be located in front of the center of projection (in contrast with Figure 2.10 where it is behind) as shown in Figure 2.12. Now, the perspective projection π is,

$$\pi : \mathbb{R}^3 \to \mathbb{R}^2, \quad \mathbf{X} \mapsto x = \pi(\mathbf{X}) = \begin{pmatrix} f\frac{X}{Z} \\ f\frac{Y}{Z} \end{pmatrix} \tag{2.42}$$

Here, $\mathbf{X}$ is the 3D point mapped to a 2D point x of the image plane. To get from 3D to 2D is a nonlinear transformation, and this nonlinearity happens because of the division with Z coordinate. As a result, in contrast with linear transformations, we can not do matrix inversions. To avoid this, at least notationaly, we introduce homogenous coordinates, taking the 3D point and adding one fourth element. With this addition, π

can be described as,

$$Z\mathbf{x} = Z\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K_f \Pi_0 \mathbf{X} \tag{2.43}$$

Now, $\mathbf{x}$ describes the homogenous coordinates of a 2D point. By having the Z coordinate be multiplied by $\mathbf{x}$, we let the remaining transformation be linear while we seperate the aspect of transformation that was nonlinear. The outcome matrices K_f and Π_0 are,

$$K_f = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \Pi_0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.44}$$

Π_0 is called the *standard projection matrix*. The job of K_f, is to scale X, Y coordinates with the focal lengths, which depend on the camera settings and are basically telling us where in the world the camera focuses. If we assume that we are far from the points of a scene, then each distance between the camera and the point is pretty much the same. This translates to the coordinate Z, the depth, being a constant $\lambda > 0$. The transformation from a 3D point to a 2D one then, happens by,

$$\lambda\mathbf{x} = K_f \Pi_0 \mathbf{X} \tag{2.45}$$

To what we saw in Section 2.4.1, where we talked about rigid body motions (an invertible matrix), we will add the idea of perspective projection to create a total model of an ideal perspective camera, that is able to make transformations from the world coordinates to image coordinates considering all this,

$$\lambda\mathbf{x} = K_f \Pi_0 \, g \, \mathbf{X_0} \tag{2.46}$$

Intrinsic Camera Matrix

Camera coordinates are such, that if i have an image, as shown in Figure 2.13, the center is usually at the bottom or top left as the pixel coordinates can not be negative. This means that if the image coordinates go from -1 to 1, the pixel coordinates go from $(0, 0)$ to $(255, 255)$. In digital cameras there is an algorithm to do the assignment of points from the image plane to pixel coordinates.

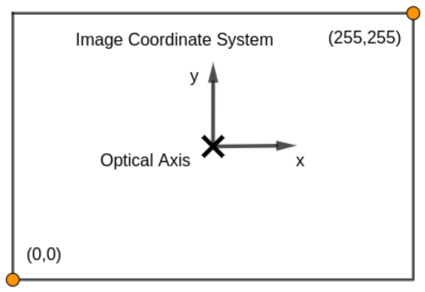


Figure 2.13: The Image Plane

Acknowledging the Equation (2.46), we will make an attempt to generalize the camera model, so that it can describe a bigger range of them. We will take into consideration that pixel coordinates may not have unit scale, and thus need two scaling factors s_x, s_y that refer to length scaling, and do the mapping from the image coordinates to pixels. Also, if the center of the camera is not perfectly aligned with the optical axis, this introduces an additional displacement for each axis, o_x, o_y, in pixel coordinates. Something that is often neglected, is that the pixles of a real camera may not be perfectly rectangular, and so introduce a certain skew factor s_ϑ (usually close to 0). In consequence, we can get the pixel coordinates $\begin{pmatrix} x & y & 1 \end{pmatrix}^\top$, regarding the homogenous camera coordinates $\begin{pmatrix} X & Y & Z & 1 \end{pmatrix}^\top$ as,

$$\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \underbrace{\begin{pmatrix} s_x & s_\vartheta & o_x \\ 0 & s_y & o_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathrm{K}_s} \underbrace{\begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\mathrm{K}_f} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.47}$$

This equation, leads us from world coordinates to camera coordinates, then from camera coordinates to image coordinates, and in the end from image coordinates to pixel coordinates. Like before, λ is the distance from camera. The matrix K,

$$K = K_s K_f = \begin{pmatrix} f s_x & f s_\vartheta & o_x \\ 0 & f s_y & o_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.48}$$

is the *Intrinsic Camera Matrix*. It is called Intrinsic, because it depends on the internal settings of the camera and not where in the world that camera is. The Equation (2.46), is then extended to,

$$\lambda \mathbf{x} = K \Pi_0 \, g \, \mathbf{X_0} \equiv \Pi \mathbf{X_0} \tag{2.49}$$

The rigid body motion g in the equation, is composed by the parameters associated with the camera motion, which are called the **extrinsic parameters**. The matrix Π is a 3×4 matrix called the *general projection matrix*. The total equation, as a transformation, may seem linear, but is in reality nonlinear because we still have the scale factor λ, which we consider constant. To get the the projection from a 3D world coordinate to pixel coordinates,

$$x = \frac{\pi_1{}^\top \mathbf{X_0}}{\pi_3{}^\top \mathbf{X_0}}, \quad y = \frac{\pi_2{}^\top \mathbf{X_0}}{\pi_3{}^\top \mathbf{X_0}}, \quad z = 1 \tag{2.50}$$

The numerator of the first equation represents the X component, while the second equation's the Y component, which are divided by the denominator Z component. The elements $\pi_1{}^\top, \pi_2{}^\top, \pi_3{}^\top \in \mathbb{R}^4$ describe the three rows of the general projection matrix Π. In general, we can face 2 scenarios of 3D reconstruction problems from images, either the camera is calibrated, and so we know K, or the camera is non-calibrated.

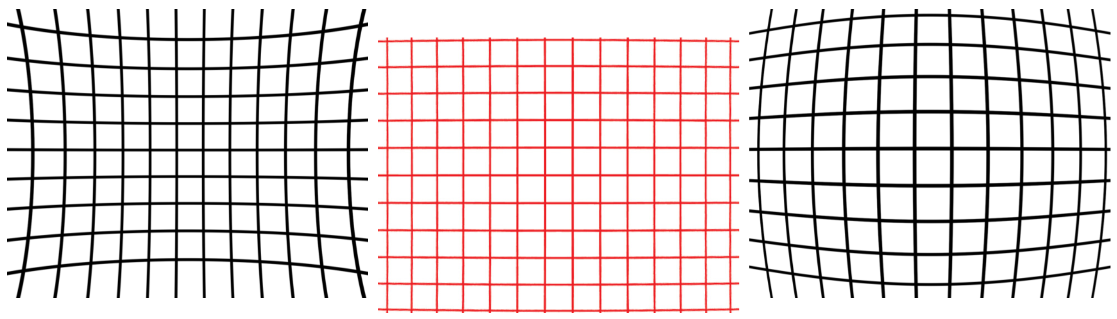
Image Distortion

Usually, camera lenses may introduce imperfections to their 2D image projection, known as distortions. These imperfections may occur due to the way in which a lens is designed, or manufacturer defects. Hence, we need to include this distortion information in our camera model to make it as efficient as possible. By considering the distortion coefficients, we can properly correct our projected 2D image. In this part of the Section, we will mention the two most common types of distortions, the *Radial distortion*, and the *Tangential distortion*. These are nonlinear distortions.

Radial Distortion We can say that all cameras have some *Radial distortion* if they use real lenses. As the name implies, the distortion increases with the increase of radius, meaning that the further from the optical center we go, the more distorted is the image. This happens because the lens bend more the light rays that are farther away from their center. The smaller the lens are, the greater the distortion that happens. An example of real-life lenses with extreme radial distortion are the fisheye lenses[1].

Figure 2.14 shows as the types of Radial Distortion. Usualy, *Pincushion* occurs in older, or low-end telephoto lesnes, while *Barrel* distortion in wide-angle lenses.

[1] https://en.wikipedia.org/wiki/Fisheye_lens



(a) Negative Radial distortion - "Pincushion"

(b) No distortion

(c) Positive Radial distortion - "Barrel"

Figure 2.14: Radial Distortion

We can describe the distortion model as an infinite series:

$$x_u = x_d(1 + a_1r^2 + a_2r^4 + a_3r^6 + \dots), \quad y_u = y_d(1 + a_1r^2 + a_2r^4 + a_3r^6 + \dots) \tag{2.51}$$

where, the point $x_d \equiv (x_d, y_d)$ is the distorted image point that is projected on the image plane via the lenses, the point $x_u \equiv (x_u, y_u)$ is the undistorted image point as projected by ideal lenses, and r is the radius of the distorted point x_d and is given by $r_d^2 = x_d^2 + y_d^2$. In case $r = 0$, there exists no distortion, while as r grows, so does distortion. Also the parameters a_1, a_2, a_3 are the radial distortion coefficients of the lenses. Usually, a_1 accounts for 90% of the distortion estimation. If the camera calibration is provided, the value of these coefficients can be found.

Another alternative was presented from Deveray and Faugeras in [8], where they suggested a method for estimating the distortion model, meaning the distortion parameters of the camera, with a set of images. In their model they considered that the distortion happens at the optical center $c \equiv (c_x, c_y)$, which may not be the center of the image. They also consider a_1 to be the first order distortion. The undistorted coordinates are given by,

$$x_u = x_d + (x_d - c_x)a_1r^2, \quad y_u = y_d + (y_d - c_y)a_1r^2 \tag{2.52}$$

Now, the distorted radius r_d is,

$$r_d = \sqrt{\left(\frac{x_d - c_x}{s_x}\right)^2 + (y_d - c_y)^2} \tag{2.53}$$

Parameter s_x represents the distorsion aspect ratio. The undistorted radius can then be computed by,

$$r_u = r_d(1 + a_1 r_d^2) \tag{2.54}$$

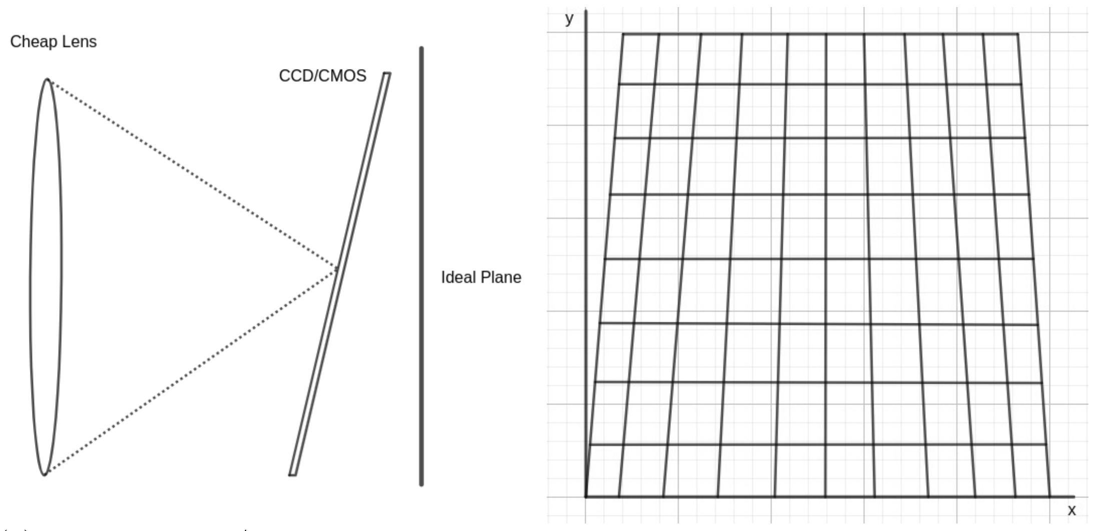
Overall, the distortion model can be mathematically formulated as a comibination of the *distortion correction factor* $f(r)$ in a fourth order expansion, and the possibly optical center displacement,

$$\mathbf{x} = c + f(r)(\mathbf{x_d} - c), \quad f(r) = 1 + a_1 r + a_2 r^2 + a_3 r^3 + a_4 r^4 \tag{2.55}$$

The distortion correction factor $f(r)$, describes how much the distortion increases as we move further away from the distortion center c. In time, new approaches appeared that could estimate the radial distortion parameters, such as the work of Stein [9] where from simple images and without knowing the camera's and points' 3D location could estimate both the radial distortion parameters and the camera's intrinsic parameters, or the work of Fitzgibbon [10] where the estimation of the lens coefficients happens simultaneously with the 3D reconstruction from the distorted images by approximating the fundamental matrix from point correspondences.

To summarize, Radial distortion causes an inward or outward point displacement on the projected image from its ideal position. The *Barrel*, as a positive Radial distortion causes outer points to gather, and *Pincushion*, as a negative Radial distortion causes the outer points to spread.

Tangential Distortion Tangential distortion is produced when the lenses are not parallel to the image plane, created by the digital camera's CCD(charge-coupled device), or CMOS(complementary metal-oxide semiconductor) image sensors. Figure 2.15a shows the way Tangential distortion is created, while Figure 2.15b the outcome of the projected image.



(a) Lens and CCD/CMOS chip placement (b) Projected image

Figure 2.15: Tangential Distortion

To model this type of distortion, we will first need the Tangential coefficients. We can describe this model isolated as,

$$x_u = x_d + (p_1(r^2 + 2x^2) + 2p_2xy), \quad y_u = y_d + (2p_1xy + p_2(r^2 + 2y^2)) \tag{2.56}$$

As with *Radial distortion* in Equation (2.52), we can further describe the Tangential distortion by considering that the distortion happens at the optical center c, which may not be the center of the image. We will provide a representation using the *Brown - Conrady model*, that is able to jointly represent the Radial and Tangential distortions,

$$\begin{aligned} x_u = x_d + &\underbrace{(x_d - c_x)(a_1r^2 + a_2r^4 + a_3r^6)}_{\textbf{Radial part}} \\ &+ \underbrace{(p_1(r^2 + 2(x_d - c_x)^2) + 2p_2(x_d - c_x)(y_d - c_y))(1 + p_3r^2 + p_4r^4 + \dots)}_{\textbf{Tangential part}} \end{aligned} \tag{2.57}$$

$$y_u = y_d + \underbrace{(y_d - c_y)(a_1r^2 + a_2r^4 + a_3r^6)}_{\textbf{Radial part}}$$

$$+ \underbrace{\left(2p_1(x_d - c_x)(y_d - c_y) + p_2(r^2 + 2(y_d - c_y)^2)\right)(1 + p_3 r^2 + p_4 r^4 + \dots)}_{\textbf{Tangential part}} \quad (2.58)$$

The parameters k_i describe the Radial distortion coefficients, while the p_i parameters, the Tangential distortion coefficients. The radius r is the same as in Equation (2.53). At last, we can define a matrix that includes all these coefficients,

$$D = [a_1, a_2, a_3, p_1, p_2, p_3, p_4] \quad (2.59)$$

This matrix D is called the *Distortion matrix*, and can differ depending on the model describing each individual camera's distortions.

2.4.3 Camera Calibration

As a camera projects a scene of the 3D world to onto the 2D image plane, there is the need to find the parameters of the camera that affect this process. *Geometric Camera Calibration* is the procedure that attempts to estimate both the intrinsic, camera's internal characteristics, and extrinsic parameters, camera's pose in the world coordinates, as seen in Equation (2.49). The *intrinsic parameters* include the focal length f and image center (u_0, v_0) measured in pixels, and the lens distortion coefficients, k_i for radial distortion and p_i for tangential distortion, which are unitless parameters, while the *extrinsic parameters* correspond to the rigid body motion, translation and rotation, of the camera.

One of the most known works on camera calibration is that of Zhang in [11], where, by having the camera constantly looking at a planar pattern provided in different orientations, without the need of knowing the camera's poses during the sequence of images, we can estimate its intrinsic and extrinsic parameters.

As we recall from Equation (2.49), the matrix Π is a 3×4 matrix called the *general projection matrix* and is given, in relation with the *intrinsic camera matrix* K and the *extrinsic parameters* g, by,

$$\Pi = K \Pi_0 \, g, \quad g = [R \mid T] \quad (2.60)$$

In general, as described from Hartley and Zisserman in [12], the point imaging equation $x = \Pi X$ is a map from world coordinates to local image coordinates. While we

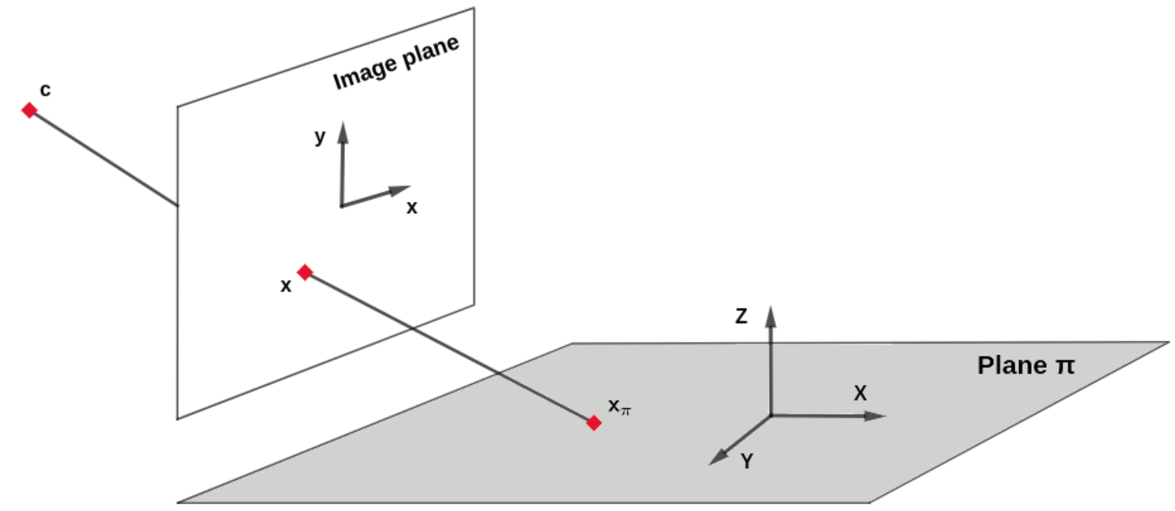


Figure 2.16: Projected points that belong in this world frame have zero Z-coordinate

decide in what world coordinate frame we are going to work, we can pick one, so that points on that plane have zero Z coordinates, as shown in Figure 2.16. We will call it *plane* π. Then, the projection from plane π to the image plane can be written as,

$$x = \Pi X = \begin{pmatrix} \pi_1 & \pi_2 & \pi_3 & \pi_4 \end{pmatrix} \begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \pi_1 & \pi_2 & \pi_4 \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \tag{2.61}$$

The parameters π_i represent the columns of the projection matrix Π. The mapping of point $x_\pi = \begin{pmatrix} X & Y & 1 \end{pmatrix}^\top$ to the image plane's point x is called a *general planar homography*, and is given by,

$$x = Hx_\pi \tag{2.62}$$

Here, H is the **Homography**, a 3×3 matrix of rank 3, which is a projective transformation that describes a non-singular linear relation between two planes. For calibrated cameras though, considering Z coordinate as zero, the projection of points onto the image plane is given by,

$$\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \underbrace{K \begin{pmatrix} r_1 & r_2 & t \end{pmatrix}}_{Homography\ tranform\ \mathbf{H}} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \tag{2.63}$$

where the r_i values correspond to the columns of the rotation matrix R. If for example the planar pattern is a checkboard, the Zhang method [11] will first need to be provided a series of images containing the checkboard, then it must locate the checkboard in each image and by finding the subpixel corners, feature points detection, of each checkboard, it estimates a homography for every image. When the homographies are generated, it makes a closed-form estimation for the intrinsic parameters of the camera from a set of them, the extrinsic parameters based on the checkboard pose, and at last the each distortion's coefficients. Note that Zhang's work considers in its model only the radial distortion parameters.

For this case, the homography defines the mapping from the planar checkboard in world coordinates, to the image plane in image coordinates. Therefore, to find the camera parameters, we need first to estimate the homography,

$$\lambda \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \tag{2.64}$$

To calculate the homography H we need at least 3 different views of the planar pattern, and because a pair of points gives us two equations, we need at lest 4 points correspondences per plane. In the end, Zhang does a total nonlinear parameter refinement with a maximum likelihood estimation conducted with the Levenberg and Marquardt algorithm, also known as damped least squares (DLS) method.

Based on the paper of Gavin in [13], the *Levenberg-Marquardt* algorithm is a nonlinear least squares method aiming to iteratively reduce the sum of the squares of errors between the raw data and the estimated ones inside a framework of updates, and can also be used on generic nonlinear least squares curve-fitting problems. This algorithm combines the advantages of *Gauss-Newton* and *gradient descent* iterative methods. First, the Gauss-Newton decreases the cost faster with every change in the attitude of the function, while gradient descent decreases the cost faster with any changes in parameter values. Levenberg-Marquardt optimization will start with a local minima estimation for the parameter values. Then, if there exists only one minima in the system, the algorithm will converge to the global minimum regardless of the initial parameter estimation, while if there exist more than one minima, the convergence to the global minimum will probably happen with a good initial parameter estimation. Overall, this algorithm doesn't always converge to the global minimum, but even if the initial guess is far from the needed, the algorithm can still converge to an optimal solution.

For example, lens distortion can be estimated by trying to minimize a nonlinear error function based on the Levenberg and Marquardt algorithm,

$$\min_{K,D,R_i,t_i} \sum_i \sum_j \| x_{ij} - \hat{x}(K, D, R_i, t_i; X_{ij}) \|^2 \tag{2.65}$$

In brief, the process is to linearize to get a function of quadratic form, calculate its derivative, then set it to zero and solve the linear system that came up. Continue with this procedure iteratively. The parameters that were obtained from the solution of the linear system can be used as initial input values to the nonlinear system.

Another work that was added in the literature, was that of Heikkila and Olli [14], where they recommended a four step calibration procedure, as an extension of the usual two step methods.

The two first steps of their model are dealing with the use of the direct linear transformation (DLT) to estimate the initial camera parameters, and then the radial and tangential distortion coefficients with the final camera parameters are estimated with nonlinear optimization for residual minimization. From their experinments, two coefficients for both radial and tangential distortion are often enough.

The third step focuses on correcting the image from assymetric projections or distortions, caused by the extraction of circular features, while the last step does a distoted image coordinates correction, by using an inverse model that interpolates the actual image point coordinates with the camera parameters that were already acquired.

Concluding, camera geometric calibration is a necessary process in computer vision when we need accurate measurements. Precise calibration can allow for difficult problems to be solved, such as reconstruction of 3D models and visual SLAM when the robot is using its cameras to navigate.

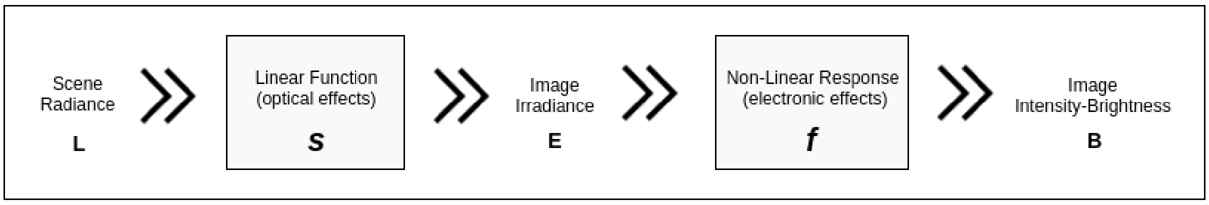

Scene Radiance
L
Linear Function (optical effects)
s
Image Irradiance
E
Non-Linear Response (electronic effects)
f
Image Intensity-Brightness
B

Figure 2.17: Mapping from scene irradiance L to image intensity B

Camera Response

A lot of vision applications are also dependant on scene radiance measurements. To describe the relation between the scene radiance and the corresponding image brightness, or intensity, we use the *camera response function*. Although this function is usually not provided for every camera setup, an estimation needs to be done to extract it. The process of estimating the inverse camera response function is called *radiometric calibration* and belongs to the family of *photometric calibrations*.

A work that strives to find a more complete space of the camera response functions is that of Grossberg and Nayar in [15]. In their paper, after collecting various camera response functions from different real-world cameras, and by analyzing each ones properties, they found common criterions every response function abides, and thus were able to create a new, low-parameter empirical model (EMoR) that can describe many other camera responses. Using this model, the response function of an unknown camera can be estimated, by recording a static scene with different exposures.

Figure 2.17 provides a complete graphical representation on how the mapping from scene radiance L to image intensity B happens. As shown, function s describes the transmission through image irradiance E, while the function f describes the conversion from image irradiance E to image intensity B. The function f is called the *camera response function* and even though it is usually nonlinear, it is spatially uniform. By using the inverse function of f, we can go from image brightness to image irradiance. We can represent a digital camera's response function in a general way as,

$$B = f(E) \tag{2.66}$$

In most cases, camera manufacturers are constructing the response function as a gamma curve. In Figure 2.18, coming from Grossberg and Nayar, we can see some of

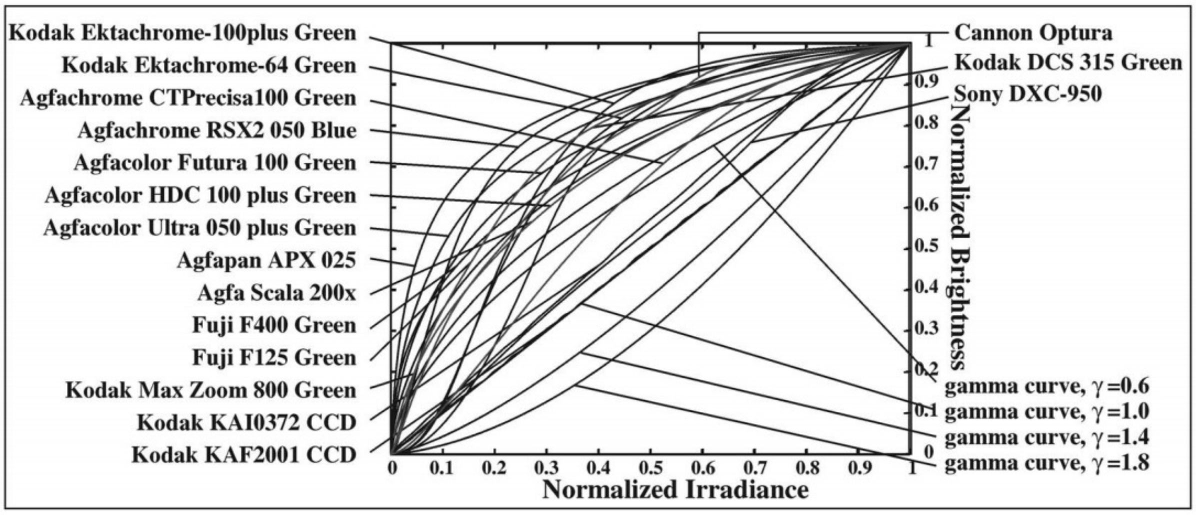


Figure 2.18: Monotonic response function examples from real-world cameras

the monotonic camera response functions that they next used as a training set for their empirical model (EMoR), with range from $0.2 \leq \gamma \leq 2.8$.

The formulation of a desired camera's response function f_G, can be done by analyzing the theoretical space of response functions W_{RF}. A linear combination of the mean response $f_0(E)$ and the basis function $h_k(E)$ can form the wanted response function as,

$$f_G(E) = f_0(E) + \sum_{k=1}^{n} c_k h_k(E) \tag{2.67}$$

where $(c_1, \cdots, c_n)$ are the coefficient parameters of the model. In other words, based on the Equation 2.67, we are trying to find a good approximation of W_{RF} that its root mean square distance, is close to that of the empirical data. To estimate the basis functions h_k and the mean response f_0, a Principal Component Analysis (PCA) is applied. And by choosing the parameters $c_k \in \mathbb{R}$, the final camera response function f_G can form. By using the empirical model (EMoR) they suggested, the approximation of the camera response function, considering $H = (h_1, \cdots, h_n)$ a basis vector, is now,

$$\tilde{f} = f_0 + Hc, \quad c = H^\top (f_G - f_0) \tag{2.68}$$

This model has also manually decides that $\tilde{f}(0) = 0$ and $\tilde{f}(255) = 255$, while the intermediate values are monotonically increasing. Along with this response function ap-

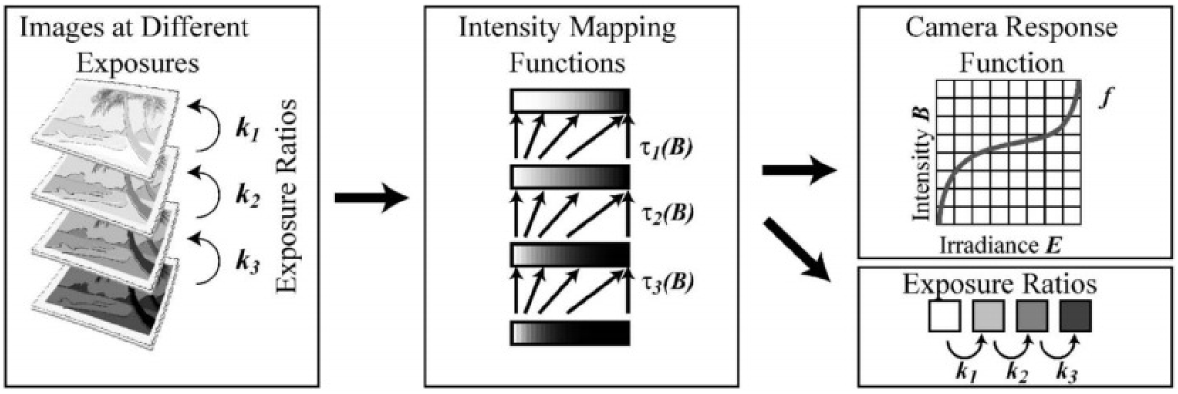


Figure 2.19: Process of recovering the camera response function and exposure ratios

proximation, comes another work of Grossberg and Nayar in [16]. Here, they suggest that the camera response can be estimated by mapping the intensity between images projecting the same scene but with different exposures, hence with an *intensity mapping function*. This way, the extraction of the response function is divided into two parts, first the intensity mapping function must be recovered, and then the response with the exposure ratios are retrieved from the intensity function. This process is presented in Figure 2.19, as shown in the work of Grossberg and Nayar in [16].

The *exposure ratio* is defined as $k = \frac{e_2}{e_1}$, where e_1, e_2 are the exposure values of two images projecting the same scene, and describes the relation between the irradiance E of the two images, as $E_1 = kE_2$. Consequently, as the irradiance needs to be estimated through the intensity, the inverse response camera function, defined as $g = f^{-1}$, is the one that needs to be recovered. And by expanding the Equation from 2.66 for two consecutive images, we have,

$$g(B_2) = kg(B_1) \tag{2.69}$$

In ideal image intensity measurement conditions, the *intensity mapping function* τ relates the intensity values between two images and can be described as,

$$B_2 = \tau(B_1) \equiv g^{-1}\left(kg(B_1)\right) \rightarrow g(\tau(B)) = kg(B) \tag{2.70}$$

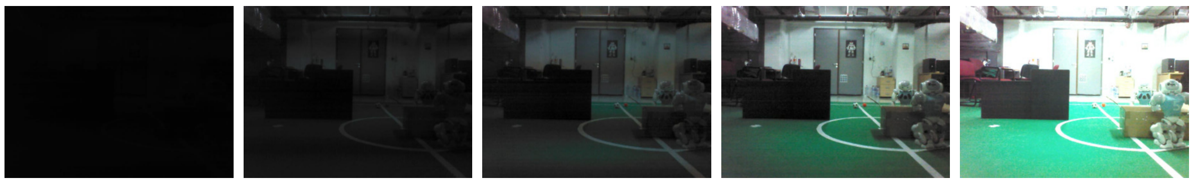


Figure 2.20: Sequence of images showing a static scene at different exposures

This mapping function between images is defined entirely by their histograms. In contrast with many other methods, by assuming that the distribution of scene radiances over a sequence of images stays close to constant, Grossberg and Nayar proved that it is possible to retrieve the intensity mapping function even with motion created by the camera, or motion in the environment, or a combination of those motions. Histograms are then used to compute the response function. An example of a sequence of images with increasing exposures projecting a static scene is shown in Figure 2.20.

Overall, recovering the inverse camera response from the intensity mapping function gives rise to ambiguity. This ambiguity can be overcomed by making assumptions about the form of the response, or making accurate exposure ratios selection for multiple images. Because with given exposure ratios there is a number of camera responses that can describe the intensity mapping, the only way to obtain a single camera response as a solution, is by making assumptions. Assumptions are also needed in case the response function and the exposure ratios are to be computed simultaneously, due to the existence of exponential ambiguity.

2.4.4 Visual Odometry

Visual odometry (VO) is the incremental estimation of a rigid-body's motion in real time using a sequence of images, an idea that was first proposed for outer space mobile robots exploration. The sensors that are used to apply VO are cameras, as it is based on visual information.

Visual odometry and visual SLAM (as seen in 2.3.3) may sound similar, but in reality they are not the same. In fact, VO is a building block for SLAM. It does not keep a long history of the camera trajectory as SLAM, but only aims at the local consistency of it, which provides it with real-time execution capabilities. And as the pose path it stores is not too big, it can a handle windowed bundle adjustment optimization.

For what visual odometry is concerned, to view and understand the motion from images, we observe the motion created from smaller parts of an image, such as lines or points. However, once we see images, we don't actually see points, but color or brightness. So, to transfer from a photometric representation to a geometric one, we can recognize a set of pixels as a uniquely characterized landmark or point, and attempt to find its corresponding location in the next image frames to associate it.

Identifying correspondence of points may seem simple at first glance, but is a big challenges in computer vision. Some of the main reasons for that, include the similarity in structure in the environment, or the difference in brightness and color of a point from frame to frame because it belongs to a shiny, reflections causing, item. Also, even if we usually assume that objects move rigidly in the environment, that doesn't stand for real life, as there exist non-rigid deformations.

In general, there are two types of Visual odometry:

- **Direct Visual Odometry**. The Direct mathod of Visual odometry is based on estimating the depth of points, which are coming from sensor measurements. They can be used to reconstruct a whole area of an environment, as they can create even dense maps of the world. Also, decisions of the model come from mostly complete information. However, they are characterized as inflexible, as they can not easily remove outliers, and also need a good initialization, as the overall model depends the visual odometry generated from the series of images.

  Direct methods consist of a single step, acquiring the sensor data, and based on the images' brightness for every pixel coordinate, they try to estimate the actual sensor values. In total, these methods aim to minimize a *photometric error*.

- **Indirect Visual Odometry**. This family of methods also leads to $feature-based$ SLAM. This type of visual SLAM is usually used for creating a sparse map of the world, as all images are sampled for features, and can contain due to efficiency only a specific number of them, based on the feature extraction algorithm used. Here, decisions are based on the not complete information of features. Feature-based methods, in most cases, are faster than direct methods, as they are more flexible, because they can seperate outliers more easily, are not greatly dependant on initialization, and are robust of system incosistencies.

They consist of two steps, first the pre-processing of the raw sensor data to provide a part of the problems solution (e.g feature correspondences), and second the estimation of the rigid body pose from the actual data, based on the first step's processed data. Indirect methods are trying to minimize a *geometric error*.

Both of these methods share the same idea for probabilistically estimating the actual sensor measurements **X**, given the noisy measurements **Y**. The goal is to find each model's parameters that maximize the probability of getting the actual measurements,

$$\mathbf{X}_{actual} = \arg\max_{X} P(Y \mid X) \tag{2.71}$$

A complementary method for Direct and Indirect visual odometry, is the addition of the robot's inertial measurements, to make the model more efficient and robust. These measurements are coming from a sensor called *Inertial Measurement Unit* (IMU), that consists of accelerometers, gyroscopes, and even magnetometers, and provides the robot with self-motion information. If an IMU's measurements are jointly contributing on the VO problem, we call the total system *Visual Inertial Odometry* (VIO). We can see an example of an IMU sensory system in Figure 2.21[1].

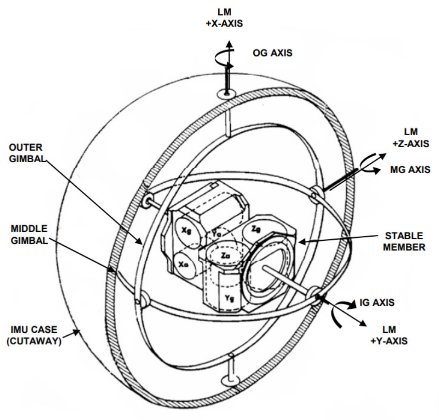


Figure 2.21: Representation of the Inertial Measurement Unit (IMU) that was used in Nasa's Apollo project

In addition, each of the two main methods, as they are able to assist the SLAM problem, can represent an environment with a great amount, or limited information. Therefore, there are two main design approaches to describe space:

[1]https://www.hq.nasa.gov/office/pao/History/alsj/

- *Dense* methods, which aim to reconstruct all pixel coordinates in an image. As each algorithm's computations include the whole image at each frame, a *geometric prior* is used to account for smothness and better pose estimation.

- *Sparse* methods, that only reconstruct a feasible and predefined amount of pixel coordinates in an image plane. Because of the distinctiveness in pixels used for every frame, these methods don't support the formulation of a *geometric prior*.

Of course, there also exist *semi-dense* methods, that combine the attributes of both the dense and sparse representation. They usually reconstruct a greater number of pixels than sparse methods, but not the whole image as the dense ones, and depending on the case, may formulate a *geometric prior*.

Optic Flow

Optic flow tries to estimate the 2D motion of points that are observable from a sequence of images, and is created by the relative motion between an observer and a scene. Optical flow is the projection of real motion onto the image plane, and is aiming to assist visual odometry (VO), which attempts to estimate a 3D body's motion. This assistance is needed, as the estimation of motion in 3D is difficult, due to the fact that the third component, the depth, is not captured in the image plane.

To create an optical flow, we need to find points correspondences. There are two cases that separate the point matching process:

- *Small deformation* in consequtive camera frames, which means that we can expect small changes at the next image. One of the most famous methods to estimate the frame displacement is the Lucas and Kanade [17] method for optical flow estimation between point features.

- *Large deformation*, which is creating large frame to frame displacements. In this case, the mapping for every pixel in a prior image to every pixel in the next image must be done. However, this procedure is computationally very expensive.

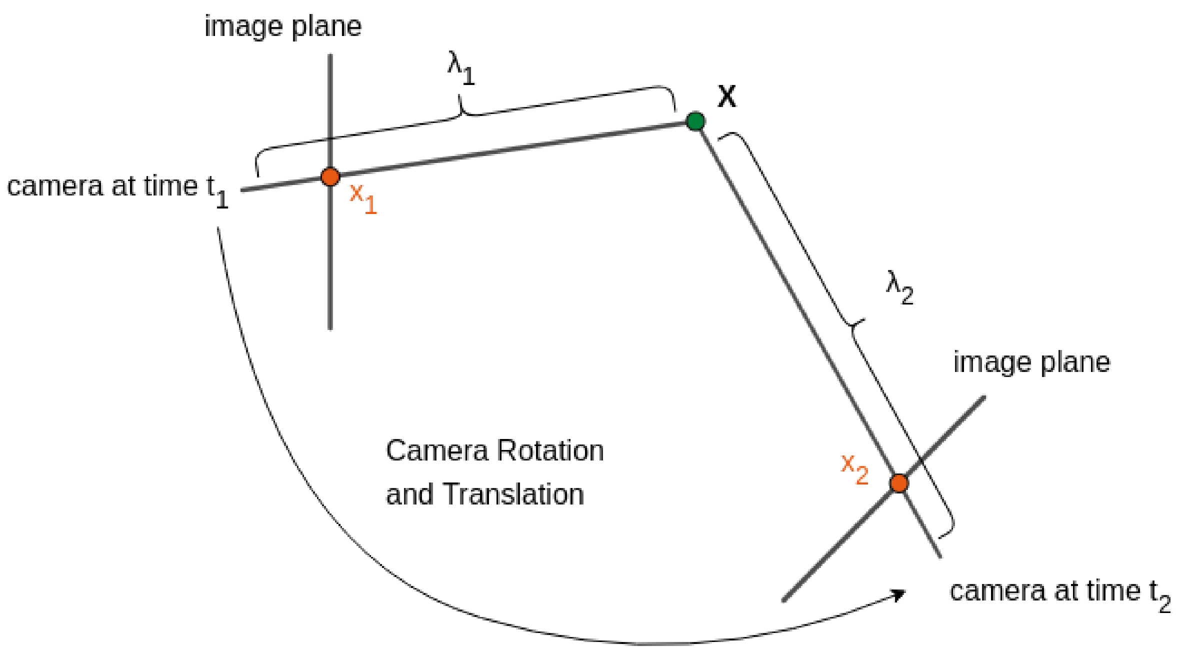


Figure 2.22: Point observation from successive camera poses

In general, it is best to assume in our problems that we deal with small deformations, as it is an easier case in contrast with having large deformations. We can then describe the tranformation of all point of a rigid body between images as,

$$x_2 = h(x_1) = \frac{1}{\lambda_2(\mathbf{X})}(R\lambda_1(\mathbf{X})x_1 + T) \tag{2.72}$$

We represent the mapping of the object from the first image to the second image with $h(x_1)$, while λ is the distance from camera, and X, x represent the point in world coordinates, and in homogenous coordinates. We can see an example in Figure 2.22. The simple relation between X and x is given with the help of the scaling factor λ by,

$$X = \lambda x \tag{2.73}$$

We can approximate the motion described above with an *Affine model* as,

$$h(x) = Ax + b \Leftrightarrow h(x) = x + u(x) \tag{2.74}$$

Here, with the affine model, which is a linear transformation, we are able to describe shrinking, divergent and skew motions. Parameter x represents the old point coordinates, and $u(x)$ the displacement or offset of point x,

$$u(x) = S(x)p = \begin{pmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{pmatrix} \begin{pmatrix} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \end{pmatrix}^\top \tag{2.75}$$

If we consider two consequtive grayscale images, **I** and **J**, and assume that a moving point's or pixel's $\mathbf{p}(t) = [x, y]^\top$ at time t, associated brigthness stays the same through the images in a static scene, then $I(p(t), t) = constant \forall t$. That means that the total time derivative is zero,

$$\frac{d}{dt} I(p(t), t) = \nabla I^\top \left(\frac{dp}{dt}\right) + \frac{\partial I}{\partial t} = 0, \quad \nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} \tag{2.76}$$

This equation is called the *differential optical flow constraint.* The vector we are interested in that describes how a point moves in the image plane is $\vec{v} = \frac{dp}{dt}$, and is called the *flow vector*, or *velocity.* The goal, if we consider $p_1 = [p_{1x}, p_{1y}]^\top$ be a point on the first image I, is to find p_2 on J, where $I(p_1)$ and $J(p_2)$ are the same point, by using the velocity $v = [v_x, v_y]^\top$ at p_1,

$$p_2 = p_1 + v = [p_{1x} + v_x, p_{1y} + v_y]^\top \tag{2.77}$$

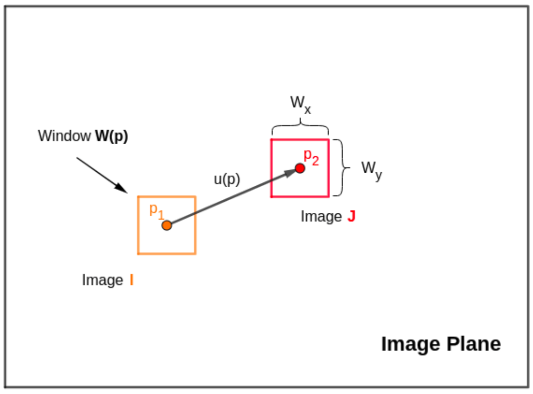


Figure 2.23: Velocity over a point displacement

To make things simpler, we assume that this velocity vector v is constant over a neighborhood or window $W(p)$ that includes point p. A Figure that graphically shows this idea is 2.23. In this figure, the window size is $W_x \times W_y$. In real life though, the brightness doesn't stay the same as we move from a camera frame to another, and certainly doesn't have the same value inside the window we just declared. In their work, Lucas and Kanade have tried to estimate the best *velocity vector* v, by using an energy function aiming to maximize the simple least squares

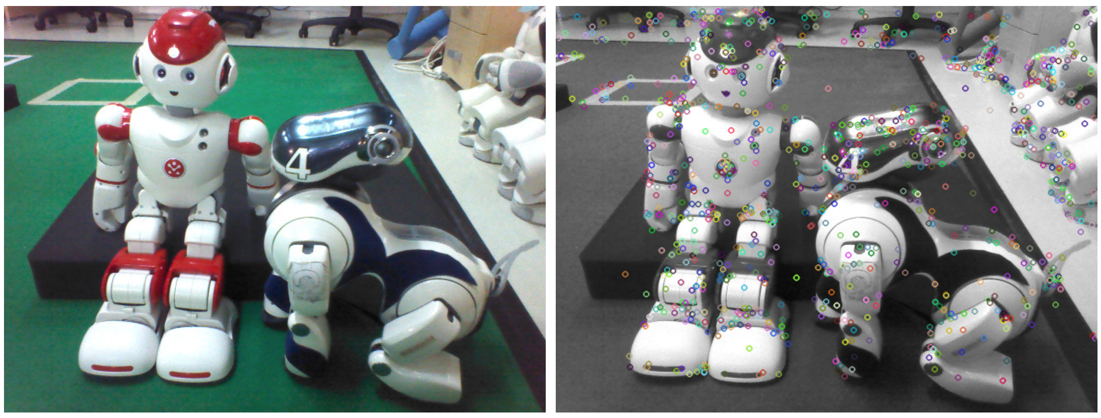


Figure 2.24: SIFT feature extraction

between the two positions of the displaced point, that can provide us with some noticeable, little movement of the window, and thus help the optical flow,

$$E(v) = E(v_x, v_y) = \sum_{W} (I(x, y) - J(x + v_x, y + v_y))^2 \tag{2.78}$$

The function E represents the difference between the earlier and next window position. To conclude, the optical flow indicates the motion of the camera and can be estimated either with feature tracking, or by tracking points that hold specific image intensities.

Feature Extraction

As we already saw, feature detection is typically used to construct an optical flow from a set of images. As features are selected in a frame, a matching at first needs to happen in the upcoming frames, that is to find the correspondence or the pair of points at each image that are the same point in the world. In brief, features can be edges, corners, or generally any distinguishable and uniquely identifiable texture in a scene. They typically don't contain a single pixel in an image frame, but a patch of them. In Figure 2.24 we provide an example of SIFT [18] feature extraction from an image.

Depending on the algorithm used, we have different *feature extractor* and *feature descriptor*. The first one decides how we pick our features from an image, while the second is responsible on how to keep and reuse the information we stored of each feature. Two of the most popular works regarding feature extraction are that of Harris and Stephens

[19], and some years later, the work of Shi and Tomasi [20].

The work of Harris was focused on a corner detector. First, window patces have to be found inside the image, that provide unique information. Then, based on the formula of Equation (2.78), and considering I and J to provide us with the intensity of the window, by working on the difference that needs to be maximized, we eventually end up with,

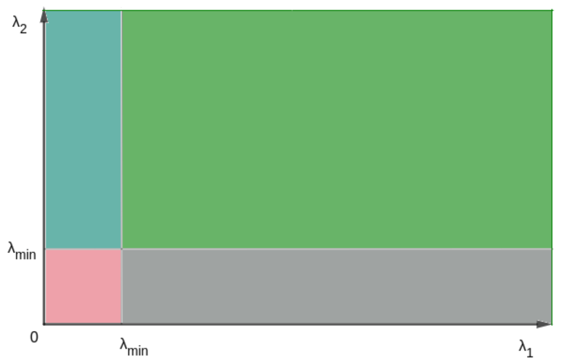


Figure 2.25: Shi and Tomasi eigenvalues interpretation

$$M(x) = \sum W(x,y) \begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix} \tag{2.79}$$

The M is called the *structure tensor*, and I_x, I_y are the partial derivatives of I. If M is not invertible and not zero, we can estimate a motion with direction indicated by the image gradient. The total energy function, in relation with the structure tensor can be written as,

$$E(v_x, v_y) \approx \begin{pmatrix} v_x & v_y \end{pmatrix} M \begin{pmatrix} v_x \\ v_y \end{pmatrix} \tag{2.80}$$

The structure tensor is of great importance in their work, as its two eigenvalues λ_1, λ_2 can determine the suitability of a determined window. A score R is therefore computed for every window,

$$R = \det M - k(\text{Trace}\, M)^2 = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{2.81}$$

If a window scores R more than an offset value, then it is treated as a *corner*, meaning a good tracking point.

Later on, Shi and Tomasi developed a corner detector that was able to provide with better results. In their work, they take also took into account a structure tensor, and decided that good points would be chosen by,

$$R = min(\lambda_1, \lambda_2) \tag{2.82}$$

In Figure 2.25 we see the way a window is defined based on its eigenvalues. When λ_1, λ_2 belong in the green zone, they are treated as *corners*. If they belong in the blue or gray, one of them doesn't meet the required value, and then the window is treated as an *edge*, while in the red zone, they don't provide any wanted information.

Summarizing, it is not always easy to decide what is a good feature. Choosing the right features in a visual odometry problem is one of the main priorities if you want a robust system, and because of the many variables that can make a feature extraction model fail, this field of study is still active and under research.

2.5 Multiple View Geometry

A single image frame from a monocular camera is not able to provide us with a lot of depth information over an object in the 3D scene, in contrast with a stereo camera setup, where the common field of view between the two singular camera units can, from the very first frames taken, provide us with a disparity map, which can lead to a 2D depth map. An example depth map created by a stereo camera from just an image taken from the left and right camera unit is seen in Figure 2.26, as presented in the OpenCV library documentation.

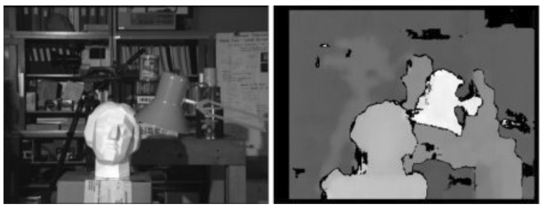


Figure 2.26: On the left we have the stereo rectified image, and on the right the estimated depth map

In order then to collect more depth data from the environment in the monocular case, we must view it from many more perspectives than just one. But because we move in the environment to observe the scene, we create the task of also estimating our 3D motion with the 3D points location. This "chicken and egg" problem we have to solve is called *3D geometry of camera and points reconstruction*. Note that the order of approaching the problem, is that we first try to approximate the camera motion, and then the location of the world points.

We will continue the discussion under the assumptions that the environment we observe is static and the camera geometric calibration has already been done to correct the projection of scene 3D world points to the 2D ones in the image plane, for a two view reconstruction that can then be generalized to multiview.

Epipolar Geometry

We can graphically see a two view problem formulation in Figure 2.27, where the extrinsic camera parameters, namely the camera's rigid body motion, need to be estimated and the X point's 3D location. The x_1, x_2 are the 2D projections of the point X in each frame respectively, and in practise the homogenous coordinates will be used for them, meaning that the third component of each matrix will be 1. Also, o_1, o_2 are the optical centers of each camera frame. The points e_1, e_2 that are created from the line that connects the two optical centers and intesects the image plane, are called *epipoles*. For two frames we have two *epipolar lines* l_1, l_2, that connect the 2D projected point of their image plane with its epipole, and for each 3D point there is one *epipolar plane*, created from the triangle (o_1, o_2, X) between the optical centers and the 3D point. The green line between the 3D points X_1, X_2 represents their correspondence after the camera motion.

We can model the rigid body motion with 6 total variables, 3 for the rotation R and 3 for the translation T, and the world point X also with 3 coordinates. The estimation of these parameters can be done even for a larger number of points i, by minimizing the following *projection error*,

$$E = \sum_i \|x_1^i - \pi(X_i)\|^2 + \|x_2^i - \pi(R, T, X_i)\|^2 \tag{2.83}$$

The lines that connect the optical center of the camera and the point in the scene are also called bundles, and estimating a close to optimal solution for them is difficult because of the complex and high dimensional space of the problem. The approximation of the best fitted bundles is done with *bundle adjustment*, a nonlinear optimization method that will be discussed in Section 2.6.

With given intrinsic camera parameters, meaning that the camera matrix $K = 1$, and distance of the point X to the optical center λ, as shown in Figure 2.22, we can represent

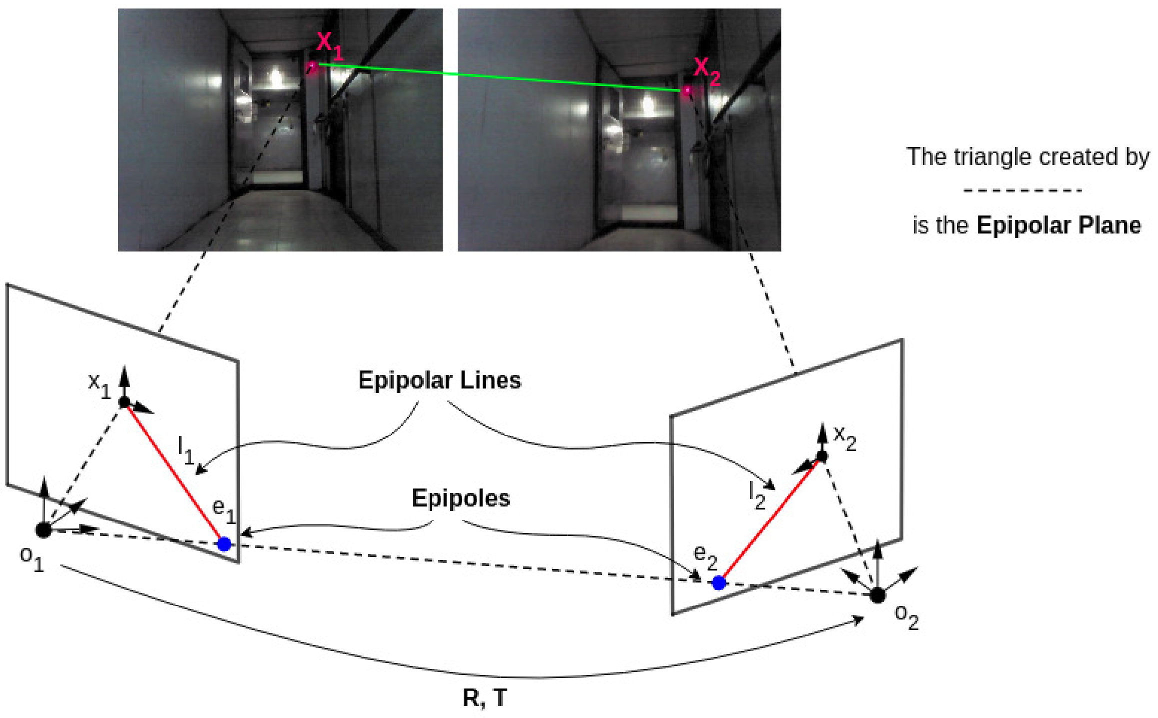


Figure 2.27: Camera motion and 3D point X depth estimation from a two view reconstruction

the overall problem as,

$$\lambda_1 x_1 = X, \quad \lambda_2 x_2 = RX + T \tag{2.84}$$

The second part of the equation contains the rotation and translation of the camera, while also the oberved depth of the point is different. By focusing to constraint this multi-variable problem, we first try to remove X from the equation, and then multiply with the 3×3 skew symmetric matrix $\hat{T}$, that models the cross product with T, we have,

$$\lambda_2 \hat{T} x_2 = \lambda_1 \hat{T} R x_1 \tag{2.85}$$

Now that the 3D point location is removed, by multiplying with $x_2^\top$ from the left we get the projection on the second frame's image plane,

$$x_2^\top \hat{T} R x_1 = 0 \tag{2.86}$$

This equation is called the *epipolar constraint*, or essential or bilinear constraint, and couples the 2D point parameters with the camera motion. Graphically speaking, it means that the bundles from the optical center do intersect in dome 3D point X, and can indeed create an epipolar plane. The matrix $E = \hat{T}R \in \mathbb{R}^{3\times3}$ contained in Equation 2.86 is called the *essential matrix*. A matrix E is called an essential matrix if it has a singular value decomposition (SVD),

$$E = U\Sigma V^\top, \quad \Sigma = diag\{\sigma, \sigma, 0\} \tag{2.87}$$

where $\sigma > 0$, and $U, V \in SO(3)$. This system has two possible solutions for $\hat{T}$ and R, but usually the wanted one is easily found as it leads to a more logical or expected point depth values.

Overall, the essential matrix can be estimated from the epipolar constraint of many point correspondences, and afterwards provide the camera's rigid body motion. The number of point pairs needed to estimate the essential matrix was proven by Longuet and Higgins[1] to be at least 8 through their *Eight Point Linear Algorithm.*

Once the camera motion is computed, we try to estimate the 3D geometry of points in the scene. We must note, that the estimated essential matrix E and camera translation T are only defined up to a scale γ, which denotes the distance between the two camera centers. In summary, the known parameters are the 2D coordinates of every point i in each camera frame, and the camera rotation R and translation T, while the unknown parameters are the relative depth of each point from the camera center λ, and the scale factor γ. The relation between those variables is seen below,

$$\lambda_2^i x_2^i = \lambda_1^i R x_1^i + \gamma T \tag{2.88}$$

[1] https://cseweb.ucsd.edu/classes/fa01/cse291/hclh/SceneReconstruction.pdf

To constraint the equation, we multiply with $\hat{x_2}^i$ from the left to remove λ_2^i, and then divide with λ_1. We can then represent it as a linear system,

$$\left(\hat{x_2}^i R x_1^i, \hat{x_2}^i T\right) \begin{pmatrix} \lambda_1^i \\ \gamma \end{pmatrix} = 0, \quad i \in (1, \ldots, n) \tag{2.89}$$

Because we want the scene reconstruction to be robust and as consistent as possible, we solve the whole problem. By considering that the vector $\vec{\lambda}$ contains all λ^i scale parameters of points and the γ unknown, we describe the whole two view system in a linear way as,

$$\mathbf{M}\vec{\lambda} = 0 \tag{2.90}$$

$$\mathbf{M} = \begin{pmatrix} \hat{x_2}^1 R x_1^1 & 0 & 0 & 0 & 0 & \hat{x_2}^1 T \\ 0 & \hat{x_2}^2 R x_1^2 & 0 & 0 & 0 & \hat{x_2}^2 T \\ 0 & 0 & \ddots & 0 & 0 & \vdots \\ 0 & 0 & 0 & \hat{x_2}^{n-1} R x_1^{n-1} & 0 & \hat{x_2}^{n-1} T \\ 0 & 0 & 0 & 0 & \hat{x_2}^n R x_1^n & \hat{x_2}^n T \end{pmatrix}$$

We try to solve this by minimizing the least squares estimate of vector $\vec{\lambda}$ described as $\|M\vec{\lambda}\|^2 = \min_{\vec{\lambda}} \lambda^\top M^\top M \lambda$. The solution comes from the eigenvector of the smallest eigenvalue of $M^\top M$.

It is important to state that the *Eight Point Algorithm*, is working only if the plane that the points are positioned on is a general 3D surface which is not planar. If the points are being located on a plane surface, then a *Four Point Algorithm* will be used. Building from where we left in Section 2.4.3, a homography H transformation is used in this case, such for a 3D point X,

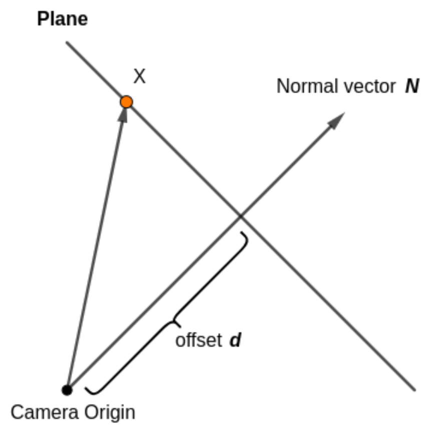
$$X_2 = \mathbf{H}X_1, \quad H = R + \frac{1}{d} T N^\top \tag{2.91}$$

The homography $H \in \mathbb{R}^{3\times3}$ is characterized by the displacement from the camera's origin d, and the normal vector of the plane N as shown in Figure 2.28. If we project the vector to point X on the normal vector N, we will get the distance d, and that's true for every point on the plane. By reformulating the epipolar constraint in Equation (2.86) and by working on the 2D coordinates of points we get,

$$\hat{x_2}\mathbf{H}x_1 = 0 \tag{2.92}$$

This equation is called the *planar homography constraint*, and by solving it we can get the motion parameters of the camera by using at least four scene points. Like with the essential matrix E, the homography H can only be approximated up to a scale factor.

Figure 2.28: Plane Representation



Uncalibrated Camera

Until now, a geometric calibration of the camera, which provides the camera matrix K as seen in Equation (2.48), was assumed. However, if a calibration is not possible to happen, and therefore the matrix K is not known, the discussed model needs to be adjusted. In general, the projected onto image plane point x' is used by any vision algorithm when it is first normalized through the given camera matrix as $x = K^{-1}x'$.

We can integrate the unknown camera matrix in our two view process by extending the epipolar constraint as,

$$x_2'^{\top} K^{-\top} \hat{T} R K^{-1} x_1' = 0 \Rightarrow x_2'^{\top} F x_1' = 0 \tag{2.93}$$

This new constraint is especially designed for uncalibrated cameras. As seen in the equation, we assume that the calibration matrix K is the same for both image frames. The matrix F is called the *fundamental matrix* and can, in relation to the essential matrix E which holds the extrinsic camera parameters, be formulated as,

$$F = K^{-\top} E K^{-1} \tag{2.94}$$

Like with the essential matrix, the fundamental matrix has a SVD as $F = U\Sigma V^{\top}$, but the singular units of $\Sigma = diag\{\sigma_1, \sigma_2, 0\}$ are not equal and not zero.

2.6 Bundle Adjustment

When a robot with a monocular camera setup has done a long course inside an environment trying to map it, drift is accumulated from its motion and its visual scale. *Bundle Adjustment* is a nonlinear optimization method that tries to simultaneously refine the 3D scene reconstruction and camera trajectory during a robot motion. In the end, it aims to provide with a jointly optimal solution for these parameters.

In real world conditions, there is noise to the projection of world point X, into the image plane point $\tilde{x}$. Therefore, the location of $\tilde{x}$ isn't equal to the true projected point's 2D coordinates, but to the noisy ones. This noise may cause the rays between different camera views to the world point X to not intersect, in contrast to what we have seen in Figure 2.27, where they do. In brief, a ray or a bundle is the line coming from the camera's optical center, passes the projected 2D noisy point $\tilde{x}$ and is directed to the possible position of the 3D world point X. Under ideal conditions these rays, assuming that the camera poses have a common field of view, intersect to the 3D points they observe.

We can represent the relation between the true 2D point position x with the camera's rigid body motion and the noisy 2D coordinates $\tilde{x}$ with the following aposteriori estimate,

$$\arg\max_{R,T,\mathbf{x}} \mathcal{P}(R,T,\mathbf{x}|\tilde{x}) = \arg\max_{R,T,\mathbf{x}} \mathcal{P}(\tilde{x}|R,T,\mathbf{x})\mathcal{P}(R,T,\mathbf{x}) \tag{2.95}$$

where $R \in SO(3)$, $T \in \mathbb{S}^2$ and together denote the motion that links two camera poses. From this formulation we can see that the probability $\mathcal{P}$ lives in the complex space of $SE(3)$. However, in real-time reconstruction problems, having the prior camera rotation and translation is not the case.

If for simplicity we consider a zero mean Gaussian noise $N(0, \sigma_i^2)$ applied to each projected point $\tilde{x}^j$ in an image, then for every image i, the Bundle Adjustment can be represented as the following cost function,

$$E(R_i, T_{i_{i=1,..,m}}, X_{j_{j=1,..,N}}) = \sum_{i=1}^{m}\sum_{j=1}^{N} \theta_{ij}|\tilde{x}_i^j - \pi(R_i, T_i, X_j)|^2 \tag{2.96}$$

As shown, this cost function tries to minimize a reprojection error, with $\pi(R_i, T_i, X_j)$ denoting the 3D projected point's, after the camera rotation and translation, 2D coordinates. Basically, by using the Bundle Adjustment method, we want to adjust the rays from m views directed to the possible observed points, in a way that they actually intersect the wanted N target points accurately. As we take into account m camera views we can decide which the first camera pose will be and usually, we set the global pose to $T_1 = 0$ and R_1 being the identity which is 1.

The parameter θ_{ij} in the equation is only to separate which points are visible in the given image and thus take part in the calculation. It can be declared as,

$$\theta_{ij} = \begin{cases} 0 & \text{if } \textit{point j not observable in image i} \\ 1 & \text{if } \textit{point j observable in image i} \end{cases} \tag{2.97}$$

This form of Bundle Adjustment can model the problem in a non-convex way. Different approaches can also represent the same problem. For example, if we take into account that $X^j = \lambda^j \times x^j$, as graphically shown in Figure 2.22, we can get a cost function that relates the noisy 2D point $\tilde{x}$ with the true 2D coordinates x and the scaling factor λ.

In order to minimize Bundle Adjustment cost functions, nonlinear optimization algorithms are used. Some of these algorithms are :

- The *Gradient Descent*, which is an iterative method that aims to compute an arbitary minimum cost function by using first-order derivatives. Depending on the location of the function it is operating, it computes its local gradient. At each iteration, it checks where does the energy decrease and chooses that direction as the preferred one.

  Assuming that the cost function is differentiable and continuous, the Gradient Descent can be applied to it. Its main advantages are that it can work on high dimensions and for any differentiable cost. However, if the cost function has many strong curves, a lot of iteration steps will need to be done.

- The *Gauss-Newton* method, that is an approximation of the *Newton* optimization. Its goal is to find an inverse hessian matrix H^{-1}, holding all second derivatives, that is guaranteed to be positive definite. In contrast to the Newton method, it is not necessary to always compute the second derivatives, meaning that if they won't probably provide us with a positive definite outcome, we drop them. The Gauss-Newton algorithm is also used for solving nonlinear non-convex least squares problems with a cost function as later shown in Equation (2.102). We can describe it by the following formulation with the iterative step being x_{t+1},

$$x_{t+1} = x_t + \Delta, \quad \Delta = -\underbrace{(\mathbf{J}^\top \mathbf{J})^{-1}}_{\mathrm{H}^{-1}} \mathbf{J}^\top r \tag{2.98}$$

The condition that decides if the Gauss-Newton method will likely work well is,

$$\left| r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right| \ll \left| \frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} \right| \tag{2.99}$$

This basically means that we have a good Hessian H approximation if the reprojection error r_i is not a big value, thus the Gauss-Newton works really well for problems that are near the linear least squares problem.

- The *Levenberg-Marquardt* algorithm. Building on what we discussed about this method in Section 2.4.3, the Levenberg-Marquardt algorithm is extremely popular for nonlinear least squares estimation. The idea behind it, is that it mixes the *Newton* and the *Gradient Descent* methods and uses them depending on the situtation we are on the cost function. The formulation that Levenberg first suggested is,

$$x_{t+1} = x_t + \Delta, \quad \Delta = -(\underbrace{\mathbf{J}^\top \mathbf{J}}_{\mathrm{H}} + \lambda \mathbf{I_n})^{-1} \mathbf{J}^\top r \tag{2.100}$$

The parameter λ decides which of the two algorithms is influencing more the total method behavior. More specifically, when λ is small, the Newton one is selected dealing with the convex part of the cost function, while when λ is a large value, the Gradient Descent is used for the non-convex part. Usually, you start with the Gradient Descent because you may be in the concave part and the Newton method is not going to work and then gradually reduce λ in the iterations.

Besides these methods, the traditional *Nonlinear Least Squares* can also be used. As a generalization of the Linear Least Squares, its goal is to find a fitting of the x inputs, which are the camera motions and 3D points, to a nonlinear model that also contains the outputs a_i, which are the 2D point coordinates generated by the mapping from the

3D ones. We can describe the reprojection error, meaning how far the projection of a 3D point is from the observed 3D point, for this model as,

$$r_i(x) = a_i - f(b_i, x) \tag{2.101}$$

Here, the function f is the projection of the 3D points to the image plane and b_i are some parameters of the projection. The cost function that needs to be minimized is then,

$$E_{lse} = \arg\min_x \sum_i r_i(x)^2 \tag{2.102}$$

In total, this is a nonlinear problem because the dependency on the unknowns x is not linear either. If we have a convex problem, the optimality condition can easily be computed from by setting the gradient to 0 as,

$$\sum_i r_i \frac{\partial r_i}{\partial x_j} = 0 \tag{2.103}$$

However, if we deal with a non-convex problem, an iterative technique as the ones discussed above, namely the Levenberg-Marquardt or Gauss-Newton methods, are more appropriate to “solve” it, or better find a locally optimal solution.

Of course, depending on the nature of the Bundle Adjustment cost function, there is a different, often nonlinear, approximation algorithm that is appropriate for each case. Overall, when the space of the environment during SLAM increases, so do the camera poses and points printed and stored in the creating map. As a result, applying Bundle Adjustment to such a large in size problem can be very costly. A way to overcome this problem is to sparsly reconstruct the environment, keeping only a small number of points and camera poses to represent the robot motion and exploration procedure. In case a more semi-dense like approach is wanted instead, it can miss out only some of the points or the frames, therefore keeping a stronger network of keyframes, or a more dense reconstruction.

Concluding, Bundle Adjustment is a nonlinear optimization process that tries to iteratively refine the camera poses and points observed in them, in a way that the points will come as possibly close to their real 3D location and thus minimize the total reprojection error. Usually, Bundle Adjustment is used only as the last step in visual reconstruction because it is time-consuming. We must also note that in order to provide accurate results, a good initialization from the robot's SLAM algorithm is required.

Chapter 3

Problem Statement

3.1 Monocular Visual SLAM for Autonomous Robot Explorations

Everyday explorations are performed by humans in simple life, or in the name of sciences, but when the area of exploration is not accessible by people, using robots is usually a viable choice to be made. These explorations' goal can vary from extraterrestrial planet mapping to deep sea aquaculture analysis, where a human can not, even with the necessary equipment, achieve this due to his physiology. However, even if he could have access to environments like these, robots could solve this problem in ways that the knowledge obtained can be easily reusable, and therefore are preferred.

Humans, and likewise robots, in order to create a map of the environment, need to solve two individual problems simultaneously, the problem of mapping and the problem of localizing themselves inside the map they create. This problem is known as Simultaneous Localization and Mapping (SLAM). If the coupled process was performed effectively, there would be little to no drift in the end map, which means that the map could then be trusted for use for other complex tasks that rely on it. Having a mapped environment and a reliable method for localization can allow a mobile robot to perform navigaiton tasks and therefore do no to less mistakes in their work.

The robots can use many sensors to autonomously map their environment, such as LiDAR laser scanners, a set of cameras, or radar sensors. When a single camera is used, we have the fully-constrained problem of Monocular Visual SLAM. Besides the drift created from the robot motion inside the environment, there is now the addition of a scale drift that accounts for the inability of a monocular camera to observe depth. In total, the monocular case is a difficult case in computer vision, but can solve the problem of SLAM with a cost-, size-, and energy-efficient sensor setup.

The monocular case is approached with Visual Odometry (VO) that tries to tell how the camera is moving based on the motion of extracted points in each individual image of a video sequence. From the camera's perspective, it can be said that the points are the ones moving. The hypothetical motion of the features is tracked and provides an Optical Flow for each one of them, which helps in the final estimate of the camera motion.

A consistent and robust system architecture must also support relocalization of the robot, meaning that, when a camera observes something that was recently seen and mapped, it can adjust the robot pose with respect to that object. It must also support loop closing, which can occur when after a long trajectory of the robot, it has returns to the starting point and tries to close the loop in its progressively created map, while correcting the motion drift accumulated in the intermediate period.

Our goal in this thesis is to implement a reliable and real-time method for Visual SLAM on our Nao robot, by exploiting the capabilities of the ROS framework. The method must be also compatible with the Nao's monocular camera setup, and cooperate effectively with a navigation module to simulate a real-life autonomous robot exploration scenario.

3.2 Related Work

Visual Odometry (VO) is divided into two main categories, the Direct VO, and the Indirect or Feature-based VO. The first tries to estimate the camera motion, while also attempting to reconstruct the 3D scene. The reconstruction is done by sampling the projected image's pixels and processing their intensity values. The Feature-based tries to achieve the same goals, but by tracking feature or landmark correspondences in a video sequence. Here, we will provide a brief overview of the most notable Indirect, Direct, and Hybrid VO systems, compatible with the monocular case. In the end, we will also mention some VO methods that use different sensor setups to supplement the space of VO systems.

3.2.1 Indirect or Feature-based Monocular Visual SLAM

In this work, we use the Indirect VO and SLAM system of Mur-Artal and Tardos [3] called ORB-SLAM in cooperation with the monocular camera setup of our Nao robot. ORB-SLAM is highly based on the PTAM [21] SLAM system, and has adopted the idea of conducting the mapping and tracking operations in two separate threads for greater efficiency, while searching for FAST feature triangulation to estimate the camera pose and points' depth. As the mapping thread is not using every camera frame in the process, it has enough time until the next keyframe is selected to enrich the last one with as many valuable points as possible. PTAM also makes use of a local Bundle Adjustment, and is used in augmented reality applications because of its good performance.

Another full visual simultaneous localization and mapping system was presented by Pirker, Ruther, and Bischof in [22] called CD SLAM. This work supports both indoor and outdoor environment exploration with a dynamic nature or with highly monotonous texture. CD SLAM also uses only a part of the total camera's frames, then creates an unweighted graph to store them and applies Bundle Adjustment to refine them. To cope with the environment's non staticity, a histogram of camera poses is embedded in each map feature's information.

The MonoSLAM algorithm of Davison and colleagues in [23] can reconstruct the 3D environment in a sparse manner. Each keypoint is assigned a depth value that is adjusted during the common camera's field of view in a probabilistic framework, and are tracked on the epipolar line. The features selected are highly informative Shi-Tomasi points and are described by their 2D location plus their computed depth. An Extended Kalman Filter is used to estimate the camera's rigid body motion through landmark correspondences in the video sequence.

A dense and indirect approach is that of Ranftl and colleagues in [24]. The objects in the scene are considered as moving, therefore a dynamic environment is modelled, and an optic flow is approximated for each to predict their movements. They are also reconstructed in the map jointly with the static environment. As moving obstacles can add a lot of noise to the camera pose estimation and therefore make the camera fail its tracking, a segmentation module was added to categorize the static from the dynamic objects, to ensure robust pose estimation and correct mapping of the environment.

Besides the ORB features [25], which are Oriented FAST and Rotated BRIEF, used in the ORB-SLAM system, two widely used feature detectors and descriptors are the SIFT [18], and the SURF [26]. SIFT features are scale, rotation, illumination, and viewpoint invariant, meaning that they offer very robust detection and tracking. First, points are extracted from an image and are represented in different scales, keypoints are selected between them with a Laplacian of Gaussian approximation, and a orientation is assigned to them after that. In contrast to SIFT, SURF features are extracted using a Hessian matrix approximation. It extends the Laplacian of Gaussian methodology by applying box filters, boosting efficiency while keeping the computations in low levels. SURF features are also scale and orientation invariant. However, because both SIFT and SURF are patended works, they can only be used in academic projects and not for commercial purposes. ORB feature detector and descriptor was created because of that reason.

3.2.2 Direct Monocular Visual Odometry

Two notable works in the space of Direct VO systems is the DTAM in [27], and the LSD-SLAM system in [28]. DTAM can create a dense map of the scene by directly processing pixels with high intensity gradient values in an image, and not by extracting features that are more informative like in the Indirect methods. During camera motion, which is described with 3 DoF for translation and 3 DoF for rotation, depths are initially estimated and then refined for specific map points to later reconstruct local scenes and objects skillfully. The camera pose trajectory is tracked in a local area through constant point alignment in a SSD optimized framework, that helps compute the inverse depth map of the scene.

A specialized method for large-scale environment explorations is the LSD-SLAM [28]. The map created from the camera's keyframes, contains the inverse depth representation and the variance of the inverse depth map, coming by reprojecting pixel intensities in new frames. The total problem brakes into three main suboperations, the tracking, the depth map estimation, and the map optimization. The map optimization is done with a pose graph optimization, similarly to the way it is done in ORB-SLAM. A loop closure happens by comparing the similarity tranformations on camera poses inside the graph, and selecting the ones that minimize a cost function.

3.2.3 Hybrid Monocular Visual SLAM

The Hybrid Visual Odometry approaches are a combination of Direct and Indirect methods, and basically aim to merge the advantages of both. One of the best performing semi-direct approaches is the SVO system in [29]. The camera initialization is done with a direct image alignment on the extracted from the image sequence points. The process is split in two threads, the first for camera tracking, done by tracking pixel intensities, and the second for mapping, using FAST feature descriptors that are quickly extracted for loop closing and localization. In the mapping thread, a probabilistic depth filter is assigned to each projected in the image feature, and is being updated with a Bayesian manner when provided more viewpoints.

A work that incorporates the information of an IMU sensor setup in the monocular Visual Odometry methodology, is the work of Li and colleagues in [30]. These type of systems are referred as Visual Inertial Odometry systems and try to increase the robustness of the SLAM process by also accounting for the measurements from the accelerometer, gyroscope, and magnetometer. It supports loop closing, and is based on the direct module for local camera tracking, and on the indirect for camera pose refinement and mapping. The features extracted are the same as in ORB-SLAM, namely ORB, and with the help of the IMU, the approximated scale that is created in the monocular case is sufficiently computed. Overall, the addition of an IMU significantly increases the efficience and accuracy of a SLAM method.

3.2.4 Stereo and RGB-D Methods

In contrast to monocular cameras, stereo setups consist of two singular camera units. These units can provide depth information for the area that covers their common field of view from the very first frames taken. Some of these works are the RSLAM system [31], the S-PTAM [32], and the LSD SLAM [33] as we previously discussed but for a stereo setup.

The RSLAM [31] is a SLAM system mainly for large-scale mapping that represents the points and camera pose in a relative manner. The system's robustness is enforced with scale invariant feature descriptors, subpixel minimization for precise tracking, quadtrees to sparsly distribute the features in the image, and by performing relative Bundle Adjustment.

The S-PTAM [32] follows the logic of the PTAM system architecture by splitting the SLAM process in two parallel threads, meaning one for mapping and one for tracking. In contrast with PTAM which uses a monocular camera, the stereo setup assists a lot in the points' depth estimation and camera pose tracking. Also, a local Bundle Adjustment runs in an individual thread to refine the covisible map. Lastly, loop correction is supported but not if the robot's trajectory was great.

The stereo LSD SLAM [33] in contrast to its monocular version in [28], computes the depth of points both from temporal and static stereo. A photometric correction is applied to refine the camera measurements and help with local map point tracking through the video sequence. The process brakes into the stereo camera tracking, the depth map estimation that initializes and filters new keyframes, and lastly a pose graph optimization to refine everything.

A special type of camera is the RGB-D setup that besides color, can provide depth information from its measurements. One of the first works in the area of mapping with an RGB-D camera is the KinectFusion method in [34]. This system supports real-time camera tracking by using the iterative closest point (ICP) algorithm and merges all depth information collected for a projected scene. The estimated scene, is represented with a dense map of hich accuracy because of the small camera motion drift. Another one is the ElasticFusion in [35] that builds a surfel-based dense map. Local and small loop closures happen in a fast pace and are then combined with global level loop closures. During this process, a non-rigid deformation is later applied that as shown, can increase the map's consistency and camera's trajectory estimation. However, this method is mostly for small environments and can not support large-scale world reconstructions.

In our work, we want to achieve robot relocalization, loop closing and effective tracking, all in real-time and with as high accuracy as possible.

Chapter 4

Our Approach

4.1 Network and ROS Communication

One of the first things that need to be established in our agent's simultaneous localization and mapping (SLAM) scenario, is the way of communicating with the robot as it navigates in the environment. This is necessary because, in order to promise both real-time computations and robust algorithm perforfance, all of the system's tasks must be managed on a different computer system than the one of the Nao robot, specified in Section 2.1, as the real-time and load requirements need to be met. More specifically, a laptop with a four-core $i7$ processor clocked at $2.4 - 3.0$ GHz, an 8 GB DDR3 RAM and a GeForce $940M$ graphics card was the main source of computing power, able to cover our needs. In addition, a wireless network communication must be maintained with max data transfer of 300 Mbps in the 2.4 GHz wi-fi band or an Ethernet connection of 10/100/1000 Mbps formed with a UTP/STP cable and received by a PCI Express bus slot.

To keep all the computational load on the remote computer station, we first installed the ROS middleware [4] on it, which was discussed in Section 2.2, and then defined it as the ROS Master, controlling all the system's topics and nodes, instead of the Nao robot. In our work, we used the Kinetic ROS distribution, as it was the one compatible with the Ubuntu 16.04 LTS operating system of the remote computer station. Figure 4.1 shows the two most compatible with our scenario network architectures. The first case is a full Ethernet connection, and enables faster data transfer and message error minimization. The second case includes an Ethernet connection up to the access point

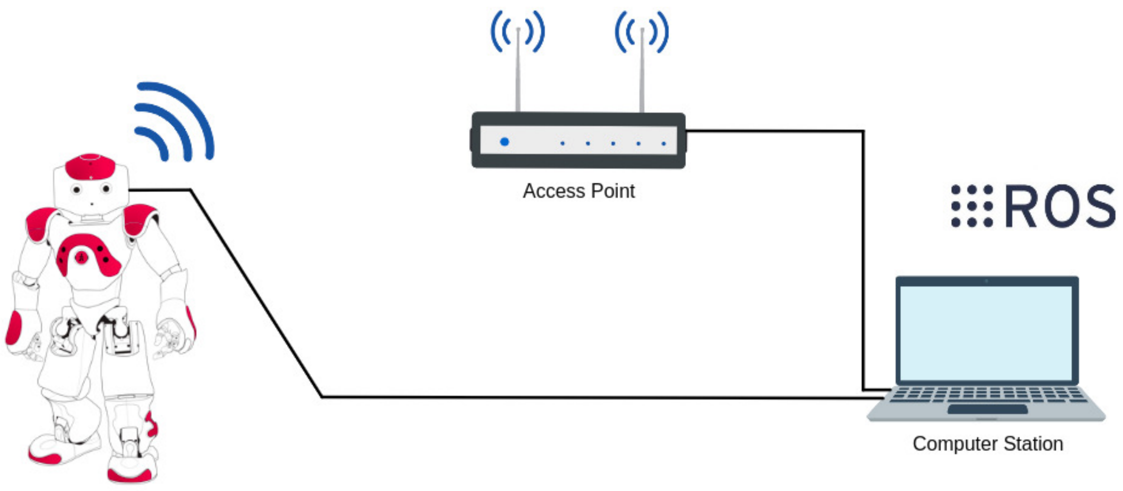


Figure 4.1: System Network Architecture

completed with a wireless LAN connection operating in the 2.4 GHz spectrum, and enables greater robot autonomy at the cost of slower communication and small-to-medium package loss depending on the distance of the robot to the router and the nature of the environment it is in.

It is notable, that for any kind of communication between the robot and the computer station, each one must know the other's *IP* address. That being said, in case of the full Ethernet connection, the robot's *IP* is obtained automatically by link-local addressing without the need of setting a static *IP* address to it, while in the case of the semi-full Ethernet connection, the router acts as the DHCP server, providing both the station and Nao with their *IP*. In our experiments, we mostly used the full Ethernet network connection for greater communication efficiency.

As the network has been established, next comes its exploitation. The *naoqi-driver* package, which is available as an open-source project [36], runs continuously in the background, bridging ROS on the computer station with the NAOqi OS on the robot, by translating the NAOqi messages to ROS messages. This way the remote computer is provided from the beginning of the experiment until its end, with real-time, organized, and adjusted to the ROS framework data, for the running processes. Figure 4.2 shows most of the *topics* and *services* created and maintained by the *naoqi-driver* node.

Additionally, since we approach the general SLAM problem by using only the robot's vision, we make some necessary changes to the camera parameters to allow for real-time performance and frame dropout minimization. Hence, instead of 4VGA (1280 × 960) resolution, we work with VGA (640 × 480) at 30 frames per second. Overall, our mobile robot contributes to the total system's computations only by sending its sensors' data to the computer station through ROS.

Topics provided through the naoqi_driver ROS node:

/diagnostics [diagnostic_msgs/DiagnosticArray]
/joint_angles [naoqi_bridge_msgs/JointAnglesWithSpeed]
/audio [naoqi_bridge_msgs/AudioBuffer]
/camera/bottom/image_raw [sensor_msgs/Image]
/bottom/camera_info [sensor_msgs/CameraInfo]
/camera/front/image_raw [sensor_msgs/Image]
/front/camera_info [sensor_msgs/CameraInfo]
/head_touch [naoqi_bridge_msgs/HeadTouch]
/imu/torso [sensor_msgs/Imu]
/odom [nav_msgs/Odometry]
/sonar/left [sensor_msgs/Range]
/sonar/right [sensor_msgs/Range]
/tf [tf2_msgs/TFMessage]

Services supported:

/camera/bottom/image_raw/compressed/set_parameters
/camera/front/image_raw/compressed/set_parameters
/get_language
/set_language

Figure 4.2: Main topics and services communicated through ROS with the *naoqi_driver* node

4.2 Robot Camera Calibration

As the Nao robot (Figure 2.2) has two non-overlapping cameras, meaning that they act as singular-monocular units and can not provide depth information from common field of view, in our SLAM scenario we have focused on using only the top camera. This camera provides around 61° field of view in the horizontal axis.

We usually define *camera calibration* as the process of determining the internal parameters of a camera, which correspond to the specific camera properties that enable the mapping from 3D world points to 2D image points, and the extrinsic parameters, which correspond to the camera's location and pose, from a sequence of uncalibrated images, as we discussed in Section 2.4.3. The significance of camera calibration as a process in computer vision applications, is seen in the difference in accuracy to the projected scene information, as in 3D representations of world structures.

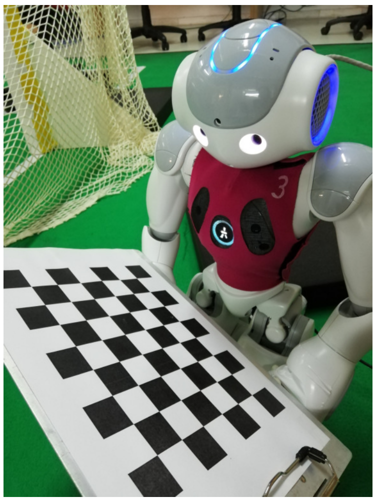


Figure 4.3: Geometric calibration process

In this work, we have conducted Nao's top camera calibration in two stages, first we estimate the intrinsic camera parameters, as described in Section 4.2.1, and then we reinforce our calibration model by also estimating the camera response function and our camera's vignette image, as described in Section 4.2.2.

4.2.1 Geometric Intrinsic Calibration

In order to acquire Nao's camera intrinsic parameters, meaning the focal length, the optical center, and the distortion coefficients, while simultaneously estimating its pose in 3D space, related as described in Equation 2.49, we used the *camera_calibration* ROS package [37]. This package provides support for both monocular and stereo cameras, and as in the work of Zhang [11], needs a known planar pattern with known dimensions, as a calibration target to be used in the process. In our case, we used a chessboard of 9×7 size, with each square's side length being $24,6$ mm. In reality though, because the calibration algorithm uses only the interior vertex points of the chessboard, the beneficial chessboard size is of 8×6. An image of the real-size robot and chess pattern can be seen in Figure 4.3.

The calibration information of a camera can be communicated inside the ROS framework with a $CameraInfo$ message, defined in Table 4.1. If the camera is uncalibrated, the matrices D, K, R, and P must be zero. Instead, if the message refers to an at least partly calibrated camera, the *Header* message, the *height* and *width* parameters, are as defined in Table 2.1, and the *distortion_model* describes the model that was used to correct the image anomalies. The matrices, D which is the distortion matrix as seen in Equation (2.59) but by using three parameters for the radial model and two for the tangential, K the intrinsic camera matrix for the raw and still distorted images as seen in Equation (2.48) but with the skew-factor as zero, R is a 3×3 rotational matrix only used for stereo cameras to align the camera with the ideal image plane, and P the projection matrix as seen in Equation (4.1).

$$\mathbf{\Pi} = \begin{pmatrix} f'_x & 0 & o'_x & T_x \\ 0 & f'_y & o'_y & T_y \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4.1}$$

The projection matrix Π holds the corrected intrinsic parameter values, including the focal length f'_x, f'_y and principal point o'_x, o'_y at the image center. The last column of the matrix is used mostly for stereo cameras, as it is related to the position of the optical center of the second camera in the first camera's frame. For us, because we deal with a monocular camera, the values of the last column are all zero. The last tree parameters of the general $CameraInfo$ message affect the geometry of the output image. The two *binning* factors represent the combination of neighborhooding pixels to super-pixels, reducing the total image resolution, and the $RegionOfInterest$ message defines the window of interest in an image, described by the a new resolution (usually set as full resolution).

The calibration node subscribes directly to the *image_raw* topic published from *naoqi_driver*, and therefore all computations happen with real-time streaming of data. If the camera has already part of its calibration information, usually provided by its manufacturer, then the node will be able to get that from the $camera\_info$ topic. In any case, when the process is over, the node can upload the computed calibration parameters to the camera driver using a *set_parameters* service call with a $CameraInfo$ message right away, or can store the calibration values for later use in a YAML file. Shortly, YAML is a data serialization language able to support the rich data structures required in today's distributed computing. YAML files are also compatible with the ROS framework, as they can cooperate with different computational units (e.g C++ or Python nodes) to retrieve and set parametes on the Parameter Server.

After starting the calibration node, in order to gain good calibration results, the chessboard needs to be waved in every direction in front of the camera, to collect information about the whole field of view. In specific, simple translations of the chessboard have to be made in the $x - axis$, the $y - axis$, simulation of *scaling* by moving towards and away from the camera, and simulation of *skew* by adding tilt with the chessboard translation. As the process is being done with a stream of data, the algorithm tries to keep track of the chessboard in the whole video sequence, and stores, for the computation of all the camera parameters, snapshots that have unique information to offer for the cases we described.

A sensor_msgs/CameraInfo ROS Message

std_msgs/Header	*header*
uint32	*height*
uint32	*width*
string	$distortion_model$
float64[ ]	D
float64[9]	K
float64[9]	R
float64[12]	P
uint32	*binning_x*
uint32	*binning_y*
sensor_msgs/RegionOfInterest	*roi*

Table 4.1: A ROS message providing meta information for a camera.

When enough snapshots have been collected, the calibration process will begin. The goal after this calibration is the continuous *rectification* of the input video sequence, meaning its correction from distortion, scaling, image plane rotation and translation. We will describe the process to end up with the wanted K and D matrices, with a single point X in the world coordinate frame as an example for simplicity:

1. First, the point will be projected in our already distorted camera model as we haven't calibrated it yet. The extrinsic parameter matrix $(R \mid t)$, containing the

combined rotation and translation of the camera in relation with a static scene, converts the 3D world point X, to a camera coordinate system's 3D point $\mathbf{X}'$,

$$\mathbf{X}' = (R \mid t)X = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \tag{4.2}$$

2. Then, the point is projected from world coordinates in the undistorted and normalized image plane's 2D coordinates in the position $\left(\frac{X}{Z}, \frac{Y}{Z}\right)$,

$$sx = \mathbf{X}' \tag{4.3}$$

3. Next, the distortion matrix D will relocate the projected point in the normalized image to its distorted position, with the help of the distortion coefficients,

$$\mathbf{x}' = D(x) \tag{4.4}$$

4. Following, the intrinsic camera matrix K will convert the earlier point coordinates to pixel coordinates (u, v),

$$q = (u, v)^\top = K\mathbf{x}' \tag{4.5}$$

5. The above four steps described the projection of point X to the original image plane. What we got from this series of transformations was the actual, distorted in the image, position of the point. In order to get the *rectified* image of the point, we must follow these transformations in the opposite order.

   In more detail, from pixel coordinates (u, v), we apply the camera matrix K and the distortion matrix D to return to the normalized and undistorted image, and then use the rotation matrix R, which is usually $R = I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, where I is the identity matrix, as there is no need to rotate the undistorted and normalized image for monocular cameras. At last, we need a new camera matrix K', with different intrinsic parameters than the previous camera matrix K, to produce the calibrated and undistorted output image. The resulted projection matrix P, as seen in Equation (4.1), projects 3D world points into the rectified image.

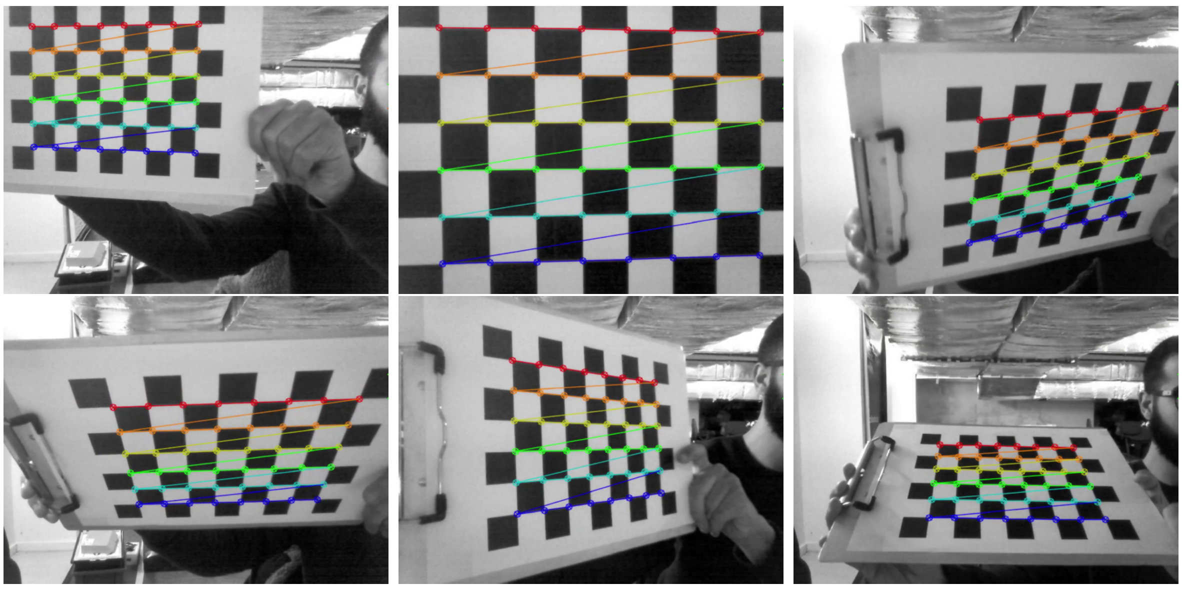


Figure 4.4: Example images of the geometric calibration process

This package uses the OpenCV *calib3d* library, that contains classes to assist some of the processes discussed in the above steps, such as the camera matrix estimation and the chessboard pattern recognition, and thus hides a lot of complexity in the software side. We provide some images during the video sequence processing from Nao in Figure 4.4.

To evaluate the accuracy of the estimated intrinsic and distortion parameters, as well as the translation and rotation matrices, a *re-projection error* is used. This error tries to estimate the normed difference between the transformed point from world coordinates to image coordinates, and the point produced by the corner finding algorithm. The average re-projection error is further computed by finding the mean of all the errors through the images used. In order to minimize that error, the Levenberg-Marquardt nonlinear optimization algorithm [13] is used, as discussed in Section 2.4.3, that computes the overall sum of squares distance between the observed in scene points, and the projected in image plane object points that have been affected by the camera parameters and pose. The re-projection error is measured in pixel units, and suggests a new estimation, for every new pattern viewed.

In our work, after varius recreations of calibration image sets, we got the best calibration results from a collection of 312 images that had the lowest mean re-projection error value of around 3%. The outcome matrices, including the camera matrix K, distortion matrix D, and projection matrix P, are shown below,

$$K = \begin{pmatrix} 555,85 & 0 & 311,14 \\ 0 & 557,91 & 235,43 \\ 0 & 0 & 1 \end{pmatrix} \qquad D = \begin{pmatrix} -0,056 \\ 0,074 \\ -0,001 \\ 0,002 \\ 0 \end{pmatrix}^{\mathrm{T}}$$

$$\Pi = \begin{pmatrix} 551,55 & 0 & 312,65 & 0 \\ 0 & 554,28 & 234,54 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

4.2.2 Monocular Photometric Calibration

After we gained good estimations for the intrinsic camera paremeters, the focus was redirected on further conducting a photometric calibration for our camera. *Photometric calibration* is the process of correcting an image in terms of pixel intensities, vignetting, auto gain or even exposure, as pixels corresponding to the same point in the 3D scene can suggest great difference in their brightness value on an upcoming image, depending on these attributes, and thus introduce errors in vision tasks. Therefore, to calibrate our camera photometrically we were based on the work of Engel and Cremers [6], were they proposed an open-source framework for manually estimating the *camera response function* and *vignette image*.

For the upcoming computations, the exposure or exposure time of every image will need to be known. In brief, *exposure* is the amount of light received from the camera sensor, and is expected to reach a pre-specified value for every image, as it is delimited by the camera apenture. *Exposure time* now, is the time needed until the camera sensor reaches the wanted exposure value, and is usually measured in milliseconds. Given the pixel value and the exposure time, we can compute the real brightness of each point in the scene. As seen in Table 4.2, the Exposure time for Nao robot, as a camera model's hardware parameters, is calculated as $\left(\frac{exposure\ value}{10}\right)$ in ms.

Parameter	Min value	Max value
Exposure	1	510
Exposure Time (ms)	0, 1	51
Gain	32	255

Table 4.2: Parameter values of localization algorithm

Nao in its default mode uses an auto exposure algorithm[1] for its cameras, that automatically changes the exposure and gain value. The algorithm subdivides every image frame, into 25 windows, that scale their exposure value with the following weighting matrix,

$$\mathbf{M}_{weights} = \begin{pmatrix} 0,25 & 0,25 & 0,25 & 0,25 & 0,25 \\ 0,25 & 0,75 & 0,75 & 0,75 & 0,25 \\ 0,25 & 0,75 & 1 & 0,75 & 0,25 \\ 0,25 & 0,75 & 0,75 & 0,75 & 0,25 \\ 0,25 & 0,25 & 0,25 & 0,25 & 0,25 \end{pmatrix} \tag{4.6}$$

Dataset Generation

Before the calibration procedure would begin, we first needed to generate each unique dataset based on our camera setup and the message material needed. For this reason, a ROS package containing the *dataset_maker* node was created to aid this task. The node subscribes only to the *image_raw* topic of Nao robot's top camera, and aims to save the incoming streaming images in a prefix name order, while also creating a *times.txt* file, with image related information, that were extracted using the NAOqi-Python API[2].

In total, after this package is used, and by considering the outcome camera parameters we computed in Section 4.2.1, the standard dataset we create for the following calibrations consists of:

- A *camera.txt* file, containing the camera pinhole model's focal length (f_x, f_y) and principal point (o_x, o_y), as well as the input image and output image width and

[1] http://doc.aldebaran.com/2-1/family/robots/video_robot.html#autoexposurealgoparam
[2] http://doc.aldebaran.com/2-4/dev/python/intro_python.html

height, which for us are the same, namely 640×480 pixel resolution. It is important to note that the values used for the focal length and pincipal point are:

$$f = \begin{pmatrix} \frac{f_x}{width} & \frac{f_y}{height} \end{pmatrix}, \quad o = \begin{pmatrix} \frac{o_x}{width} & \frac{o_y}{height} \end{pmatrix} \tag{4.7}$$

as in the software side, these values are considered in relation to the image width and height. Also, a value to specify the rectification mode for the image exists. This parameter can crop the image to the biggest possible rectangular, keep the full field of view but paint undefined image regions as black (this mode is usually used for debugging), rectify the image based on the camera parameters that are given, or do nothing.

- A compressed *Images file*, containing the dataset's images that the calibration algorithms will be applied to. The max number of images, their color code (RGB or grayscale), and their file format(png or jpg), are all defined in the ROS launch file of the package.
- A *times.txt* file, that has stored specific information about every image in the dataset. In more detail, it contains information about the id, Unix timestamp, and exposure time of every image frame.

Camera Response Function Estimation

Using the package we just discussed in 4.2.2, we create a standard dataset from a sequence of 1000 images by recording a static and rich in shades scene, while smoothly changing the camera exposure from its min, to its max value as defined in Table 4.2. The overall procedure is seen in Figure 4.5. Influenced by Engel and Cremers in [6], when irradiance is unknown, the camera response function G and lens attenuation V, can only be examined as scalar factors. The formula that can jointly model these parameters, while still considering the exposure time t, and the image irradiance B is,

$$I(x) = G\left(tV(x)B(x)\right) \tag{4.8}$$

Here, I is an image, and $I(x)$ denotes a pixel's, in the image coordinates, intensity. In order to isolate the lens attenuation V from the above equation to focus on estimating

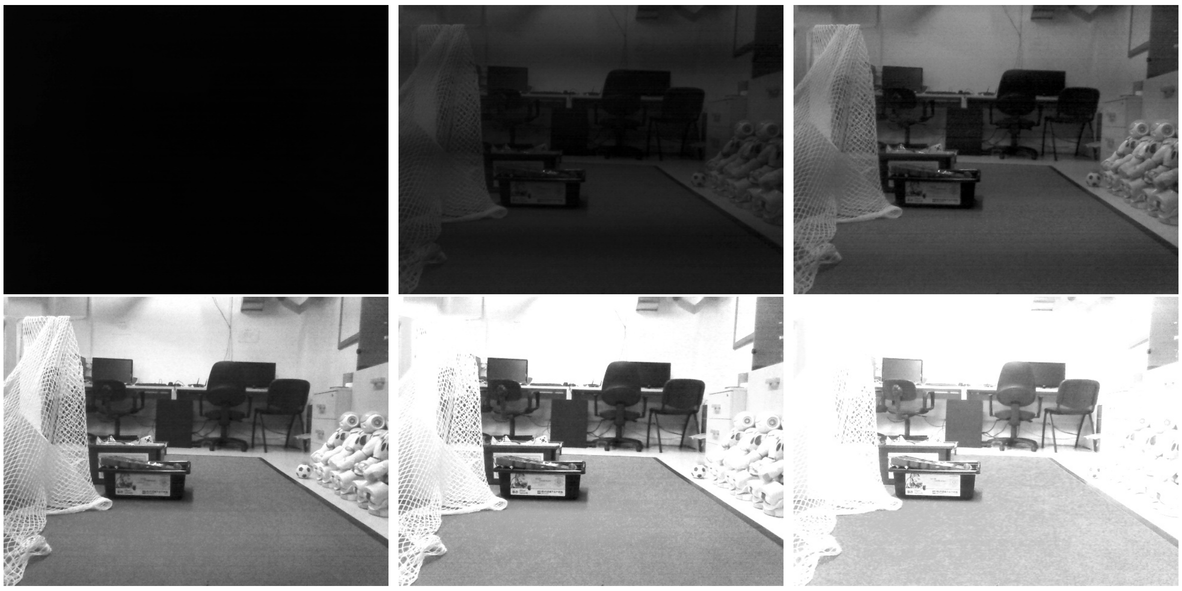


Figure 4.5: A small portion of the dataset's images used for calibration (in grayscale), showing a static scene with increasing exposures from left to right and from top to bottom

the camera response function G, we express it as a factor of the image irradiance, $B'(x) \equiv V(x)B(x)$, and thus,

$$I(x) = G\left(tB'(x)\right) \tag{4.9}$$

By taking into account all the images i in the dataset that have different and increasing exposure time value, we can describe the total cost function E with a Maximum Likelihood formulation as,

$$E = \sum_{i} \sum_{x \in \Omega} \left(U(I_i(x)) - t_i B'(x)\right)^2 \tag{4.10}$$

To estimate E, we have assumed a Gaussian white noise $U(I_i(x))$, where U represents the inverse response function G^{-1}. The inverse U between two images, is basically based on their gray values and their exposure ratio, and logarithms are used to generate a linear algorithm that can determine it. In Figure 4.6 we can see examples of the estimated logarithmic image irradiance $\log(B')$. Also, if Ω_k describes the family of pixels in the

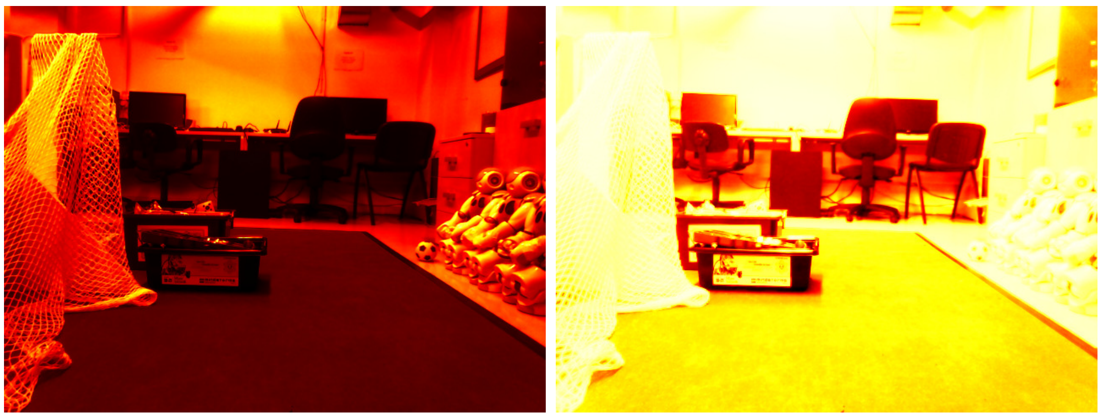


Figure 4.6: Estimation of the log image irradiance between an image with small exposure value (left) and big exposure value (right)

whole dataset that have their intensity $I(x)$ equal to k, then the minimization of U and $B'(x)$ becomes as,

$$U(k)^* = \arg\min_{U(k)} E(U, B') = \frac{\sum_{\Omega_k} t_i B'(x)}{\mid \Omega_k \mid} \tag{4.11}$$

$$B'(x)^* = \arg\min_{B'(x)} E(U, B') = \frac{\sum_i t_i U(I_i(x))}{\sum_i t_i^2} \tag{4.12}$$

The procedure of estimating the inverse camera response function U, is also called *radiometric calibration*. It is notable that overexposed values, such as $U(255)$, are removed from the estimation, and are manually replaced as $U(255) = 255$ and $U(0) = 0$. From the resulting inverse camera function U, we can return to computing the original camera function G. In Figure 4.7 we can see in short how U ended to its last function representation through the dataset. It is a 256×256 matrix, with its x-axis being the intensity value of the pixel $I(x)$, and y-axis the irradiance $U(I(x))$.

The outcome of this calibration, with the lowest mean re-projection error value of around 11%, is the estimated inverse camera response function U, and is stored in a *pcalib.txt* file containing a single row of 256 increasing distribution of values. Each value, describes a map from the $(0, \ldots, 255)$ spectrum to the respective irradiance value, which is proportionate to the discretized inverse response function.

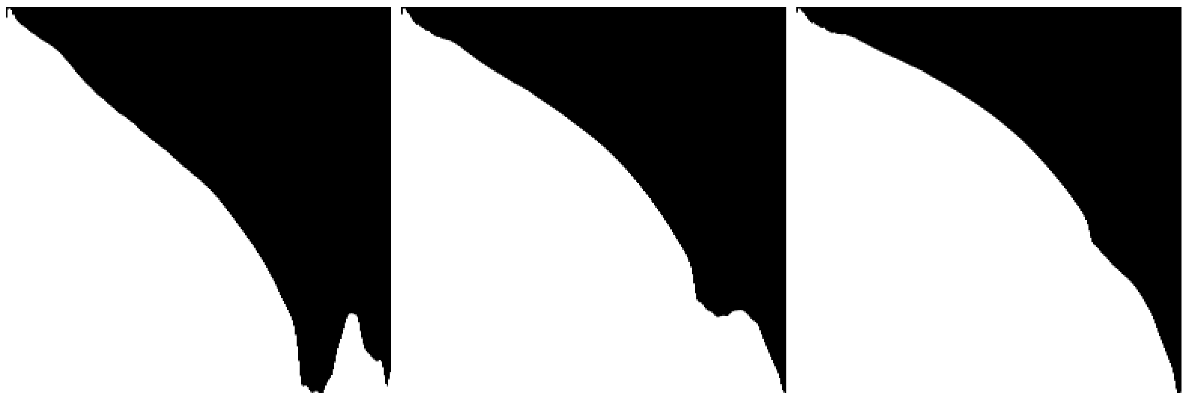


Figure 4.7: Evolution of the inverse camera response function estimation during the dataset with enabled gamma correction from left to right. In the x-axis is the pixel intensity value, and in the y-axis the irradiance

Vignetting Removal

To make the non-parametric vignette calibration on our camera setup, just like with the camera response function calibration, we first need to create a dataset. For that purpose, we create a standard dataset based on the ROS package discussed in 4.2.2, with 600 images by recording a smooth in brightness Lambertian scene, meaning a scene of diffusely reflecting background surface, with a planar pattern from different camera perspectives. The exposure is also set to a static value of 333, meaning 33.3 ms that is mentioned for all images in the *times.txt* file. In contrast with the *camera.txt* file of the camera function calibration, the rectification mode we use now is *crop* instead of *none*.

The planar pattern used for us, was an AR Marker, as seen in Figure 4.9 from the work of Jurado, Salinas, and more in [38]. In this paper they presented a fiducial marker system, which can provide the ability for various marker dictionaries generation in size and number of bits, individual marker detection with error correction, and occlusion detection with a method for overcoming the problem by using a color map to compute the occlusion mask.

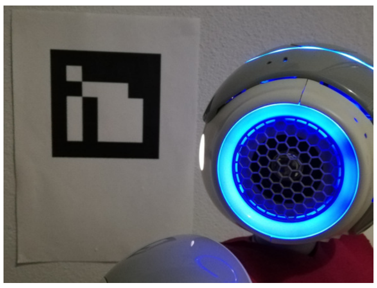


Figure 4.8: Vignetting Removal process

In brief, to detect a marker in an image, the image is first converted to grayscale and an image segmentation is then applied in search for useful contours. The image contours are then extracted and filtered to external the marker's outer shape. The process that follows is the marker code detection, as every marker represents a different *id* represented an integer value that is encoded into a grid of black and white pixels. The inner region is thus analyzed by first computing the marker's homography to extract the contour information, which will be decoded in binary $0, 1$.

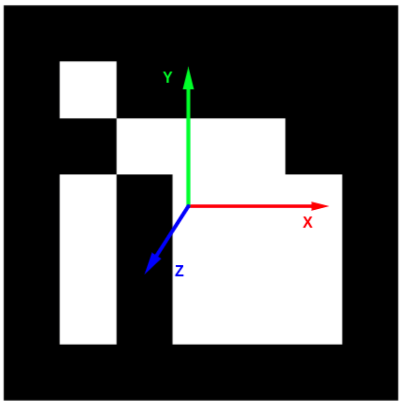


Figure 4.9: The AR Marker used with pose

Afterwards, the marker's *id* needs to be identified, as a lot of candidates have been generated that can describe it. In case that there was no match inside the dictionary, a correction is needed to aid the task by calculating the distance of each candidate to all other markers in the dictionary. The Hamming algorithm is used to calculate the distance between two markers m_i, m_j, as,

$$D(m_i, m_j) = \min_{k \in 0,1,2,3} \{H\left(m_i, R_k(m_j)\right)\} \tag{4.13}$$

where H defines the sum of hamming distances between each pair of markers, and R_k is a function that can rotate clockwise the marker for $90°$. We can further represent the distance of a marker to its closest one in a dictionary as,

$$D(m_i, \Delta) = \min_{m_j \in \Delta} \{D(m_i, m_j)\} \tag{4.14}$$

Lastly, once the marker has been identified, a corner refinement is applied and then a pose estimation is achieved by using the Levenberg-Marquardt nonlinear optimization algorithm [13] to minimize the re-projection error. The main benefit of these markers is that their detection is robust, simple and fast.

In our work, we are able to use these highly reliable markers for locating the marker, decoding it, and estimating its 3D pose, namely position and orientation in space, through the ArUco Library [39]. Each marker in the library is represented by a 2D grid of black

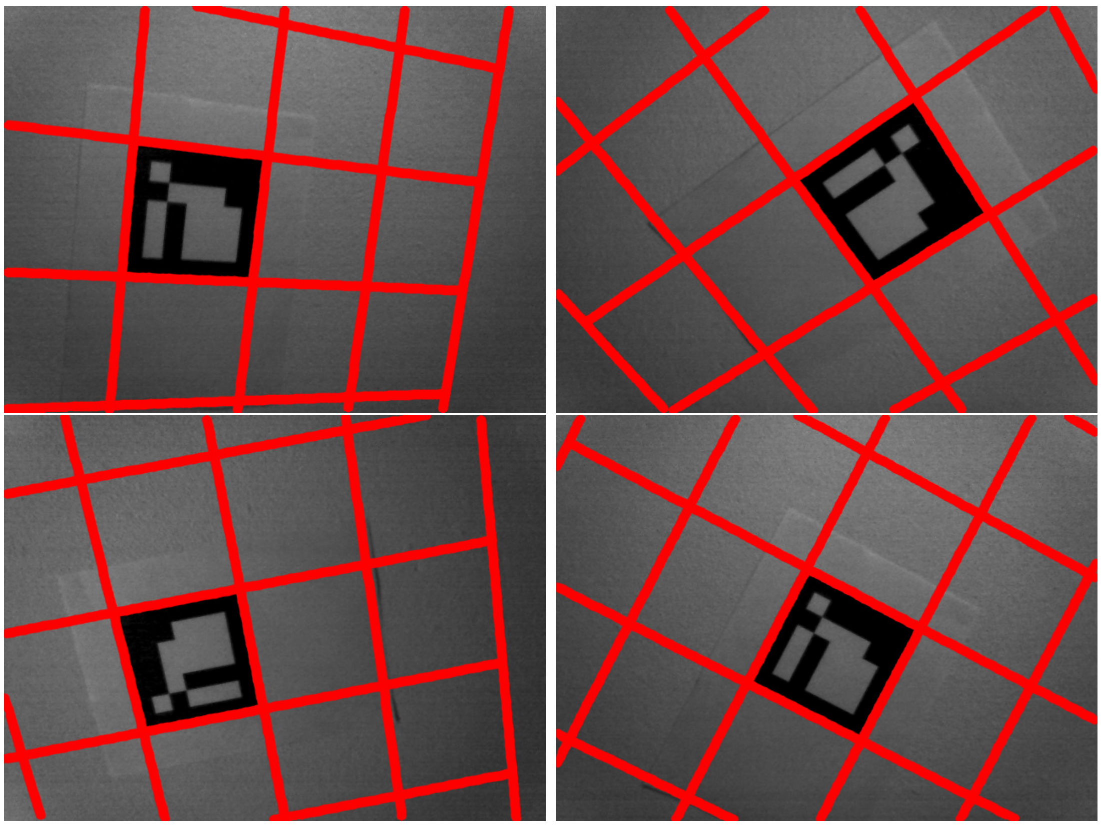


Figure 4.10: A small portion of the dataset's images used for vignette calibration, overlaid with a 3D plane P in red

and white pixels, and thus holds binary information. The marker we used in Figure 4.9 consists of a 5×5 grid and has an *id* of 213. In the grid, the second with the fourth column represent the data bits, while the first, third and fifth column the parity bits. Because there are 5 values included in each column , the total number of markers that could be encoded based on the data bits are $2^{10} = 1024$ markers.

In addition to the standard dataset created for this calibration, we have to provide the generated *pcalib.txt* file containing the inverse camera response function, of Section 4.2.2. We can see the procedure of recording the scene containing the AR Marker in Figure 4.10. The camera pose, with respect to the marker pose, is estimated by using a planar surface P that is seen in red in the referenced figures. By describing the projection of a point from the planar surface P to a pixel in the image as π, and by again assuming Gaussian

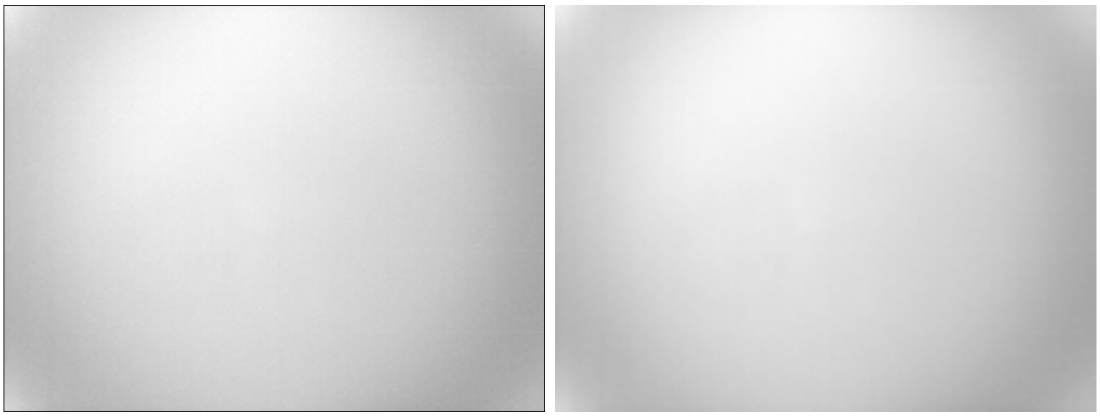

(a) Original Vignette map (b) Smoothed Vignette map

Figure 4.11: Estimated attenuation factors V as monochrome 16-bit image

white noise $U(I_i(\pi_i(x)))$, we are led to the Maximum Likelihood energy formulation,

$$E = \sum_{i,x\in P} \left(U(I_i(\pi_i(x))) - t_i V(\pi_i(x)) C(x)\right)^2 \tag{4.15}$$

Here, C denotes the planar surface irradiance. The minimization of the energy function E leads to the minimization of irradiance C and attenuation factor V as,

$$C(x)^* = \arg\min_{C(x)} E(C,V) = \frac{\sum_i t_i V(\pi_i(x)) U(I_i(\pi_i(x)))}{\sum_i \left(t_i V(\pi_i(x))\right)^2} \tag{4.16}$$

$$V(x)^* = \arg\min_{V(x)} E(C,V) = \frac{\sum_i t_i C(x) U(I_i(\pi_i(x)))}{\sum_i \left(t_i C(x)\right)^2} \tag{4.17}$$

The resulting vignette map is normalized, so that its pixels' max value is 1 and its min value is 0. The outcome of this calibration process is a monochrome 16-bit png image describing the calibrated vignette function as pixelwise attenuation factors and is seen in Figure 4.11. A smoothed version of the vignette function, which is able to correct intense pixel values fluctuations and black borders, is also provided. Also, in Figure 4.12 we can see the estimated irradiance image C, depending on the plane P, from our dataset.

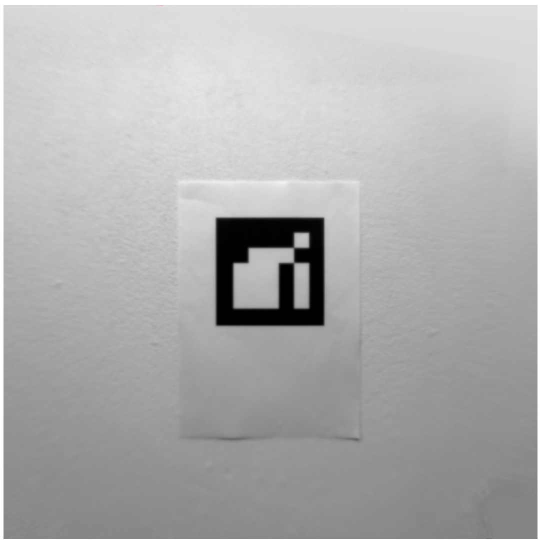


Figure 4.12: The estimated irradiance image C given the 3D plane P

4.3 Inverse Depth Estimation

The monocular camera can provide no to less depth information of a point in the world scene from a single view, and can only estimate it at a scale through subsequent measurements. Altogether, the more viewpoints are used to observe an object, the more information we can collect about its depth and location in space. We call the angle created by two viewpoints on a point, that point's *parallax angle*. As obvious, the farther away a point is from the camera, the smaller the value of its parallax angle even if we create a lot of translation with our robot's camera, which leads to the problem of initializing that point's depth.

The parallax contributes directly to the point's depth estimation, and enough parallax needs to be created to compute it. Therefore, an assisting method is needed for a more efficient depth approximation. The work of Civera, Davison, and Montiel in [40] is widely used in our system, providing with an inverse depth parametrization of points. This work is an extension of their previously published papers in [41], [42], and can now maintain a large number of points represented in inverse depth while the needed computations happen in real-time. As discussed in Section 4.2, a camera calibration is assumed, as it will highly benefit the robustness of the whole procedure.

In contrast to other approaches, initialization is supported for scene points, both close and far from the camera. The inverse depth value is initially assigned to each point directly at the time it was first observed, and through tracking, it is corrected to reach an approximately "optimal" value, within the standard extended Kalman filter (EKF) framework.

The total state vector, containing the camera information x_u and every point y_i observed at each frame, is,

$$x = \left(x_u^\top, y_1^\top, y_2^\top, \ldots, y_n^\top\right)^\top \tag{4.18}$$

We can briefly describe the camera state at each frame by its location r^{WC}, its orientation with a quaternion q^{WC}, its velocity v^M and angular velocity ω^M in a vector,

$$x_u = \begin{pmatrix} r^{WC} \\ q^{WC} \\ v^M \\ \omega^M \end{pmatrix} \tag{4.19}$$

where C denotes the camera's coordinate frame, and W the world's coordinate frame. Each point coded in inverse depth is described by 6 in total parameters, and is given by,

$$y_i = \begin{pmatrix} x_i & y_i & z_i & \theta_i & \phi_i & \rho_i \end{pmatrix}^\top \tag{4.20}$$

The vector $\begin{pmatrix} x_i & y_i & z_i \end{pmatrix}^\top$ contains the camera's optical center position at the location where the point was first observed, θ_i and ϕ_i are the azimuth and elevation angles based on the world coordinate frame that define the vector m, and ρ_i the point's inverse depth which is inversely proportional to the semi-infinite ray d_i, and equals to $\frac{1}{d_i}$.

The main drawback of representing a point with its inverse depth, is the accumulation of computations, as each point is represented with 6 parameters, making the state vector grow a lot larger. As shown in Figure 4.13, we can jump from the inverse depth representation to the standard Euclidean one $x_i = \begin{pmatrix} X_i & Y_i & Z_i \end{pmatrix}^\top$, as,

$$\begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} + \frac{1}{\rho_i}\mathbf{m}(\theta_i, \phi_i) \tag{4.21}$$

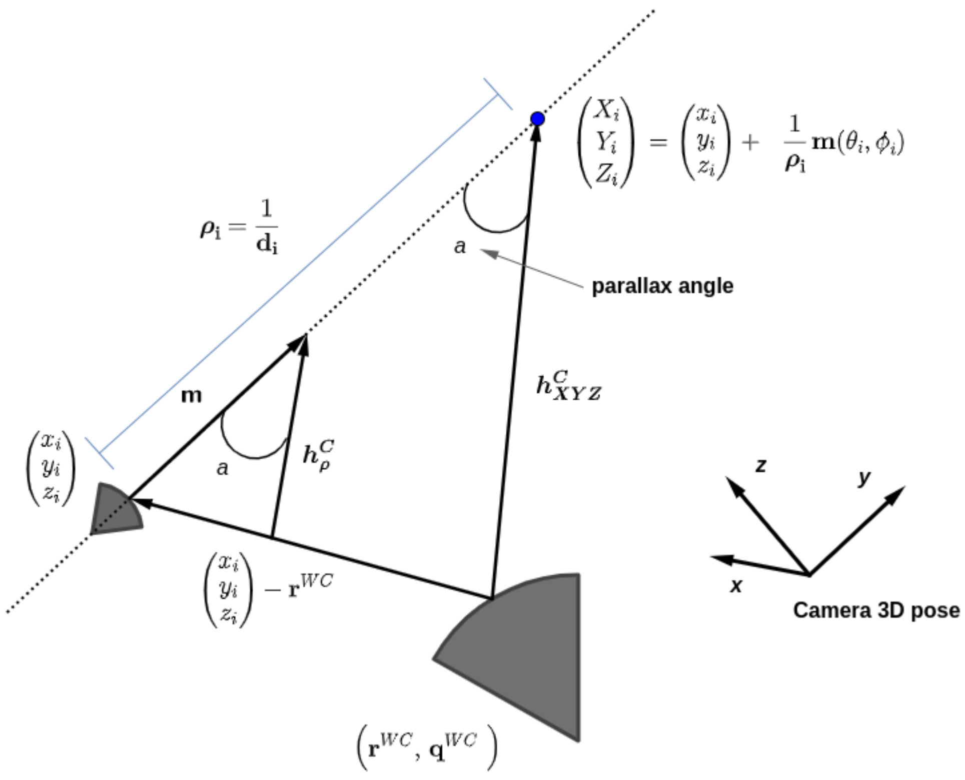

$\begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} + \frac{1}{\rho_i}\mathbf{m}(\theta_i, \phi_i)$
$\rho_i = \frac{1}{d_i}$
a
parallax angle
m
h^C_{XYZ}
$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$
a
h^C_ρ
z
y
x
Camera 3D pose
$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} - \mathbf{r}^{WC}$
$(\mathbf{r}^{WC}, \mathbf{q}^{WC})$

Figure 4.13: Point inverse depth parametrization

where,

$$\mathbf{m} = (\cos(\phi_i)\sin(\theta_i) - \sin(\phi_i), \cos(\phi_i)\cos(\theta_i))^\top \tag{4.22}$$

The relation in Equation (4.21) transforms a 6D point into a 3D one. When the geometric calibration parameters, namely the focal length, principal point, and pixel size for image width and height are applied to the incoming camera measurement, a ray can be accurately traced as a vector h^C from the camera's principal point to the Euclidean represented or inverse depth parametrized point,

$$h^C_{XYZ} = \begin{pmatrix} h_x \\ h_y \\ h_z \end{pmatrix} = R^{CW}\left(\begin{pmatrix} X_i \\ Y_i \\ Z_i \end{pmatrix} - r^{WC}\right) \tag{4.23}$$

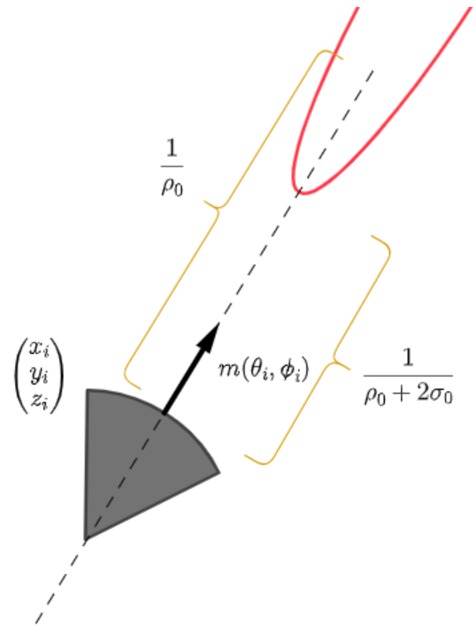

$\frac{1}{\rho_0}$
$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix}$
$m(\theta_i, \phi_i)$
$\frac{1}{\rho_0 + 2\sigma_0}$

Figure 4.14: New point observation

$$h_\rho^C = R^{CW} \left(\rho_i \left(\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} - r^{WC} \right) + m(\theta_i, \phi_i) \right) \tag{4.24}$$

As seen from these two equations, the outcome vector h^C is analogous to the camera location, the point's location, and the camera's rotation matrix R. The parallax is created by the $\left(\rho_i \left(\begin{pmatrix} x_i & y_i & z_i \end{pmatrix}^\top - r^{WC} \right)\right)$. For distant from the camera points, where the parallax and their inverse depth ρ_i is close to zero, the ray can be estimated by a shrinkage of Equation (4.24), as $h_\rho^C \approx R^{CW}\left(m(\theta_i, \phi_i)\right)$. These points can't offer any beneficial information about the camera's translation, but keeping them can still be useful, as they can be used to estimate the camera's rotation.

When a point is observed, all parameters of the vector (4.20) can be described except its inverse depth ρ_i. The depth prior, modelled by a Gaussian distribution, covers a big region because of the uncertainty as shown in Figure 4.14 with red. The covariance σ_ρ and ρ_0 are also initialized and create the inverse parametrization interval, able to represent depths to infinity but not at zero, of the point as shown in Figure 4.15.

When the depth estimation of a point is satisfactory, the conversion from inverse depth parametrization to the XYZ system can happen with a *linearity index*, that detects the appropriate time for the conversion to happen, so that both efficiency, as the computational issue of maintaining a 6D point will no longer exist, and robustness are achieved. The directinal ray from the camera to the point, as transformed by the world coordinate frame, is given by,

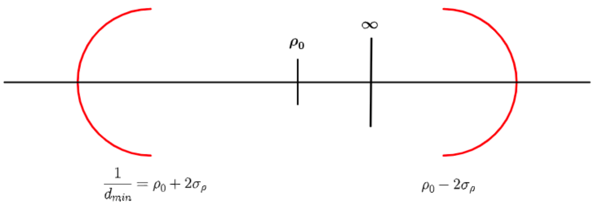


Figure 4.15: Point's confidence region

$$h^W = R^{CW}\left(\mathbf{q}^{WC}\right)\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{4.25}$$

where the matrix $\begin{pmatrix} u & v & 1 \end{pmatrix}^\top$ represents the observed point's location in pixel coordinates after the calibration correction. The azimuth θ_i, and the elevation ϕ_i angles that affect the ray's direction can be computed as,

$$\begin{pmatrix} \theta_i \\ \phi_i \end{pmatrix} = \begin{pmatrix} \arctan(h_x^W, h_z^W) \\ \arctan\left(-h_y^W, \sqrt{h_x^{W2} + h_z^{W2}}\right) \end{pmatrix} \tag{4.26}$$

Summarizing, this work enables our monocular SLAM system to represent points at all depths, with low or high parallax, with an inverse depth parametrization that can dynamically change to the standard Euclidean when needed, while the computations can happen in real-time. A total behavioral example is given, as provided by the paper, in Figure 4.16.

4.4 Direct Monocular Visual Odometry

Building on what we discussed in Section 2.4.4 about Visual Odometry, the *Direct* approaches aim to estimate the camera motion, considered as a rigid body motion, by sampling pixel intensities over a sequence of images. In our work, we used the Direct Sparse Odometry (DSO) system of Engel, Cremers, and Koltun in [2]. As its name implies, this system tries to approximate a monocular camera's motion through direct visual

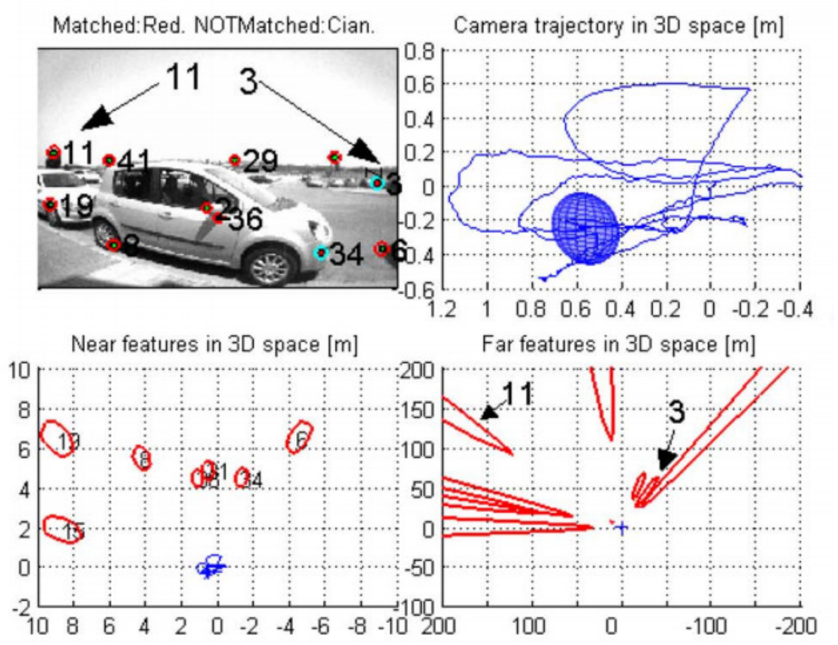


Figure 4.16: Inverse depth parametrization example on close and far from the camera points

odometry, and sparsly represent the environment with a point cloud map. Everything that follows up, is by considering our camera's model to be the simple pinhole model.

Because direct visual odometry is extremely sensitive to image deformations and continuous intensities variation, besides the geometric calibration of the camera presented in Section 4.2.1, that aims to repair the projection from a 3D point in the scene to a 2D point in the image, a photometric calibration is also necessary as presented in Section 4.2.2, to correct the mapping from image irradiance to pixel intensity.

The photometric calibration brakes into two parts as modelled in Equation (4.8), that of estimating the camera response function, which is a nonlinear function mapping from $\mathbb{R} \rightarrow [0, 255]$, and that of extracting the lens attenuation factors that map from $\Omega \rightarrow [0, 1]$. The combined correction of this calibration is applied at every image frame of the real-time video taken by Nao as,

$$I(x)^* = \frac{G^{-1}(I(x))}{V(x)} \tag{4.27}$$

where $I(x)^*$ denotes the corrected image. The camera response function we used is seen at the right of Figure 4.7, and the lens attenuation but in its smoothed version in Figure 4.11b. To evaluate the correction of our photometric calibration over a point $\mathbf{p} \in \Omega$ that was first projected into the image plane and then had its 2D position adjusted, we use,

$$E_{pj} = \sum_{p \in N_P} w_P \left\| I_j[p'] - b_j - \frac{t_j e^{a_j}}{t_i e^{a_i}} (I_i[p] - b_i) \right\|_\gamma \tag{4.28}$$

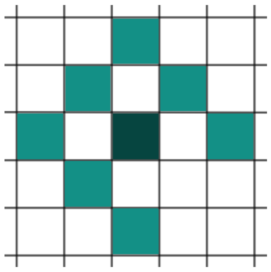


Figure 4.17: Pattern of pixels used over every point p_i

Here, the photometric error E_{pj} of the picked point p in the image, is computed as the gaussian-weighted sum of squared differences (SSD) of a small neighboring area of pixels consisting the point p, referred as N_P. Figure 4.17 shows the pattern of pixels that was used. This structure was especially selected, because it allows for optimized processing with each one of the 8 Streaming SIMD Extension (SSE) registers, from $xmm0$ to $xmm7$, taking care of each one of the 8 pixels of interest of each point pattern, and thus greatly increase CPU performance. Also, there is no need to compute every pixel in the point's neighborhood and slow computations, as the pattern selected provides satisfactory information of the total point's intensity.

Continuing, t_i and t_j denote the exposure times, in the images I_i and I_j that the point was observed in, while w_P is a gradient weight that aims to give less influencing power on values coming from pixels with high gradient. The point p', as a point coming from the calibrated image, contains information about the camera's rigid body motion as seen in Section 2.4.1, and the point's estimated inverse depth value d_p as discussed in Section 4.3. Its relation to these parameters can be represented by the following function,

$$p' = \Pi_c \left(\mathbf{R} \Pi_c^{-1}(p, d_p) + \mathbf{t} \right) \tag{4.29}$$

where Π_c is the projection of the point from the 3D scene to the 2D image with c being the geometric calibration parameters, namely the focal length and the principal point that we have already computed, and $\mathbf{R}$ with $\mathbf{t}$ are the camera's rotation and translation. In addition, the *Huber norm* $\| * \|_\gamma$ is applied in Equation (4.28). It was chosen because even with a small number of erroneous observations the analysis is not ruined, as only a significant number of them can make a change on the outcome. The observations far from the x-axis are given less weight, while those close to the center are more capable of influencing the analysis. Huber loss is in total less sensitive to outliers than the mean squared error loss (MSE).

We can further compute the total photometric error $E_{photometric}$ of the whole video recorded by our robot's camera, by considering the photometric error E_{pj} in every frame F, for every point extracted in that frame p, and for every corresponding frame that the point p was seen in, as,

$$E_{photometric} = \sum_{i \in F} \sum_{p \in P_i} \sum_{j \in obs(p)} E_{pj} \tag{4.30}$$

Figure 4.18 provides a graphical representation of the discussed process of observing the environment through a sequence of images. In this case as an example, we have four keyframes, meaning four different camera poses in the 3D world, that observe five different points. The blue lines from each camera's pose, define the host frame that each point belongs to, while the red line defines the observance of a point from a different pose than that of the host's. With green, we can see the camera's trajectory. Each point, after being examined from the different perspectives it was seen, will be allocated with an inverse depth value d_p depending on the last keyframe it was tracked from.

The new reprojection error then, is computed by also considering every frame's, that has observed the point and can provide information about its depth, pose. Moreover, as the exposure time is known at each frame and a photometric calibration is available for our model, a new prior can be added to minimize the affine brightness transfer function,

$$E_{prior} = \sum_{i \in F} (\lambda_a a_i^2 + \lambda_b b_i^2) \tag{4.31}$$

Here, a_i and b_i are the parameters of the brightness transfer function, that every keyframe in Figure 4.18 has. Overall, in this direct approach, each point is characterized only by its inverse depth value and hence there is only a single unknown to be estimated.

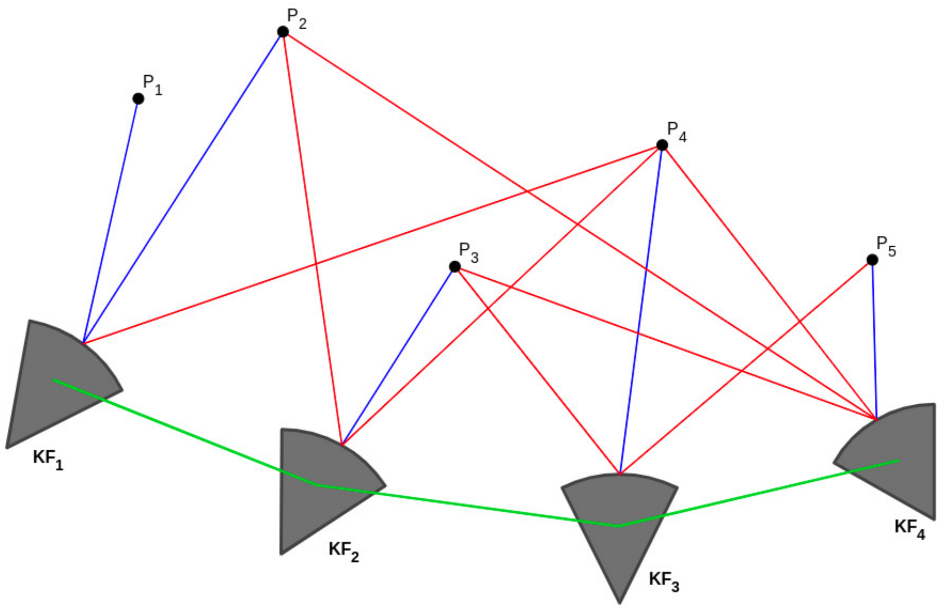


Figure 4.18: An example Factor Graph for our direct odometry model

System Variables Optimization

The total photometric error $E_{photometric}$ of Equation (4.30) is minimized by using the Gauss-Newton optimization in a sliding window with,

$$H = J^{\top} W J, \quad b = -J^{\top} W r \tag{4.32}$$

Here, J is the Jacobian matrix of the stacked residual vector r, used to optimize the system's variables. Also, W is a diagonal matrix that contains the weights used for the residuals, and H is the Hessian, organizing all second partial derivatives into its matrix, and containing the correlations between depth values.

Now, J_k denotes the Jacobian of a single pixel p_k contained in the 8 in total residual pixels of point p and is given by,

$$J_k = \frac{\partial p_k \left((\delta + x) \boxplus \zeta_0 \right)}{\partial \delta} \tag{4.33}$$

The state vector $\zeta \in SE(3)$ represents all the optimized variables of the system, which consist of the camera poses, camera intinsic parameters, points' inverse depth, and affine brightness parameters, and is given by $\zeta = x \boxplus \zeta_0$, where x belongs to the Lie algebra of rigid body poses as discussed in Section 2.4.1. The symbol $\boxplus$ describes a simple addition for all system variables, except camera poses where it maps from $se(3) \times SE(3) \mapsto SE(3)$. The relation between Lie algebra and Lie group is seen in Figure 2.9. After analysis, the Jacobian J_k can be broken down to,

$$J_k = [J_I \; J_{geo} \; J_{photo}] \tag{4.34}$$

where J_I concerns derivatives of the image gradient, J_{geo} of the geometric parameters, and J_{photo} of the photometric ones. The last two Jacobians J_{geo} and J_{photo} are first estimated using the First Estimate Jacobian method. Then J_{geo} is assumed to be constant for all of the point's residual pixels and thus is evaluated only for the center pixel providing real-time computation enpowerment. on the whole, for every keyframe created, 6 Gauss-Newton iterations are done.

When a lot of old variables still exist and new ones need to be estimated, a marginalization is taking place using the Schur complement in order to keep the structure of the Hessian sparse.

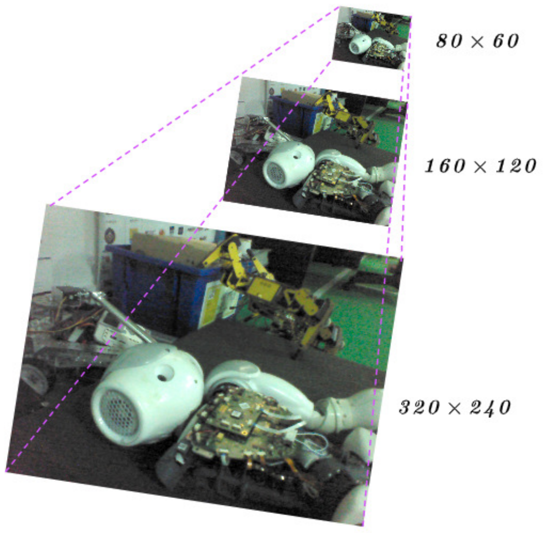


Figure 4.19: Pyramid levels created in our case

Keyframe Selection

We can break the DSO system's real-time visual odometry processing into two main parts, the *keyframes management* and the *points management* of every keyframe. The *keyframes management* can be broken down to the following steps:

1. **Frame Initialization**. Between all the incoming frames or images that Nao robot's camera generates and sends to the DSO system, which are around 25 frames per second decreased from the starting value of 30 mostly due to network, only a small number of them are considered keyframes. When a new keyframe is tracked, then the points will be analyzed in the image and will produce a semi-dense depth

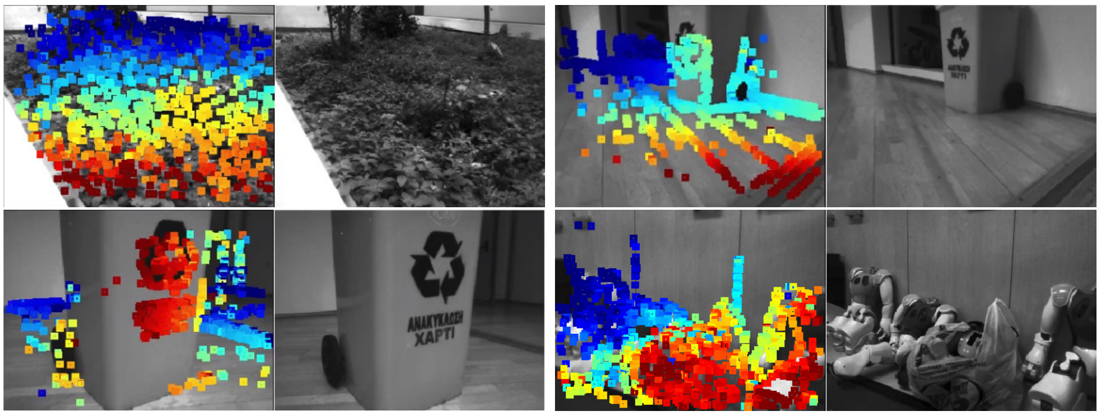


Figure 4.20: Example depth maps coupled with their initial images. From red to blue we denote the close to far points

map. Then, every next incoming frame from Nao, will be tracked in reference with the previously chosen keyframe using a consequtive frame image alignment, and a multi-scale pyramid applied on the new frame with the number of levels based on the image resolution. In Figure 4.20 we provide some example depth maps taken in real-time from our Nao robot. Red color refers to close points, while blue to far ones.

For the down-scaled in the pyramid images, a pixel can be given a depth value only if one of the pixels in the previous pyramid level that correspond to this pixel had a depth value. Also, the root mean squared error (RMSE) is used to measure the image alignment success until the last computed frame, and if its value is double the RMSE until the previous frame then a retracking needs to be done to save the whole process, by using a matrix that produces 27 different rotations on the top pyramid level generated, and therefore with the smallest resolution. In our case the pyramid that is created has three levels as shown in Figure 4.19, with the coarsest resolution being 80×60 and the initial, after downsizing, 320×240.

2. **Keyframe Extraction**. From the approximately 25 frames received, around 10 of them are decided to be keyframes, while the others are discarded.

   A new keyframe is in total created, first because of a significant camera pose change, which is translated to an optical flow created from the selected points between the last pack of keyframe-frame, and is measured by a mean square as

$\left(\frac{1}{n}\sum_{i=0}^{n}\|\mathbf{p}-\mathbf{p}'\|^2\right)^{\frac{1}{2}}$. A keyframe is also added because of great exposure time changes, decided by a brightness factor computed from $|log(e^{a_j-a_i}t_jt_i^{-1})|$, or image occlusions and disocclusions that can turn a point in the image to disapear when it was visible and vice versa. The last is computed as in the case of the camera's pose change, but with the difference that the camera rotation is not taken into account and is considered as the identity I, and thus $\left(\frac{1}{n}\sum_{i=0}^{n}\|\mathbf{p}-\mathbf{p'_t}\|^2\right)^{\frac{1}{2}}$.

In addition to the previous reasons of creating a keyframe, an extra one that considers all of them at the same time exists, that assigns a weight on each one, and if their sum is greater than a prefix value, then a keyframe is produced.

3. **Keyframe Marginalization**. It is notable that we keep on real time 7 active keyframes during our direct odometry process. From the set of all active keyframes each second, a number of them will be marginalized in order to keep a trackable camera trajectory.

   In more detail, if less than 5% of the points in a keyframe $KF_i \in$ in the set of active keyframes, are visible in the last keyframe, then the keyframe KF_i is marginalized. Also, if for some reason the number of active keyframes is more than 7, then the surplus frame which maximizes a heuristic function's distance score, from the latest keyframe, is marginalized too. The marginalization process first is applied on the keyframe's points and then to itself, and can lead to almost half of the keyframe's points to be discarded. Even if it seems as not a very efficient method, it helps on the energy function optimization.

Point Selection

In contrast with other direct approaches, this work mainly focuses on sub-sampling the images to allow for real-time executions in a jointly non-linear optimized framework. After analysing the keyframes management, we will further show the steps done for the *points management* at each keyframe:

1. **Candidate Point Extraction**. At each keyframe, points are selected based on their high image gradient in relation to their surrounding pixels, and their good distribution on the image plane. In general, we try to keep a fixed number of 2000 active points between all existing active keyframes. A point's gradient is measured

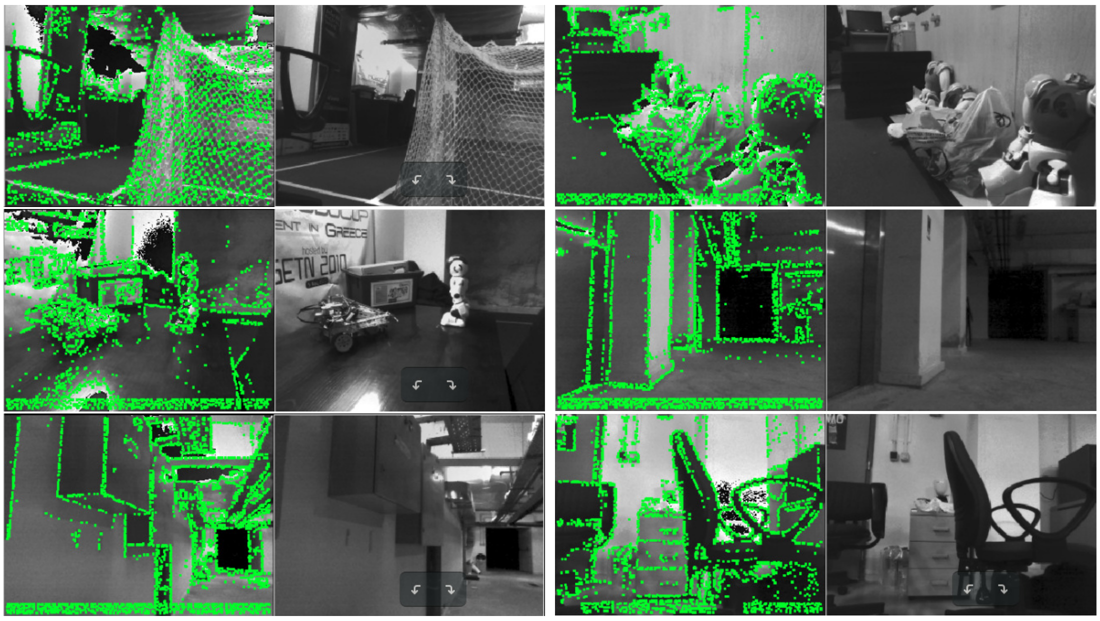


Figure 4.21: Example candidate point selections (shown in green) taken from real-time Nao images

by first splitting the image into 32×32 pixel areas, and then computing a gradient threshold g_{th} for each block that is mostly based on the mean block's gradient value. To then beneficially distribute the points in the image, we split the image into $d \times d$ blocks, and pick from each block, the pixel that has gradient value larger than its block's threshold g_{th}, and the highest of that from every other pixel in the chosen block.

Additionally, in case of low textured patterns such as white walls, where not many gradient variations exist and thus it would be difficult with this method to extract points from that surface, the gradient threshold g_{th} is dynamically reduced and the discussed procedure is repreated for new block sizes of $2d$ and $4d$. The value of d is also dynamically adjusted so that the overall value of 2000 active points between all active keyframes to be valid. However, specifically for the initiallization, there are chosen 2000 points in each keyframe but only as candidates. Figure 4.21 provides some examples on candidate point selections taken in real-time by our Nao robot.

2. **Point Tracking**. The selected candidate points are not instantly used in the framework process. From their set that belongs to a keyframe, a search is being

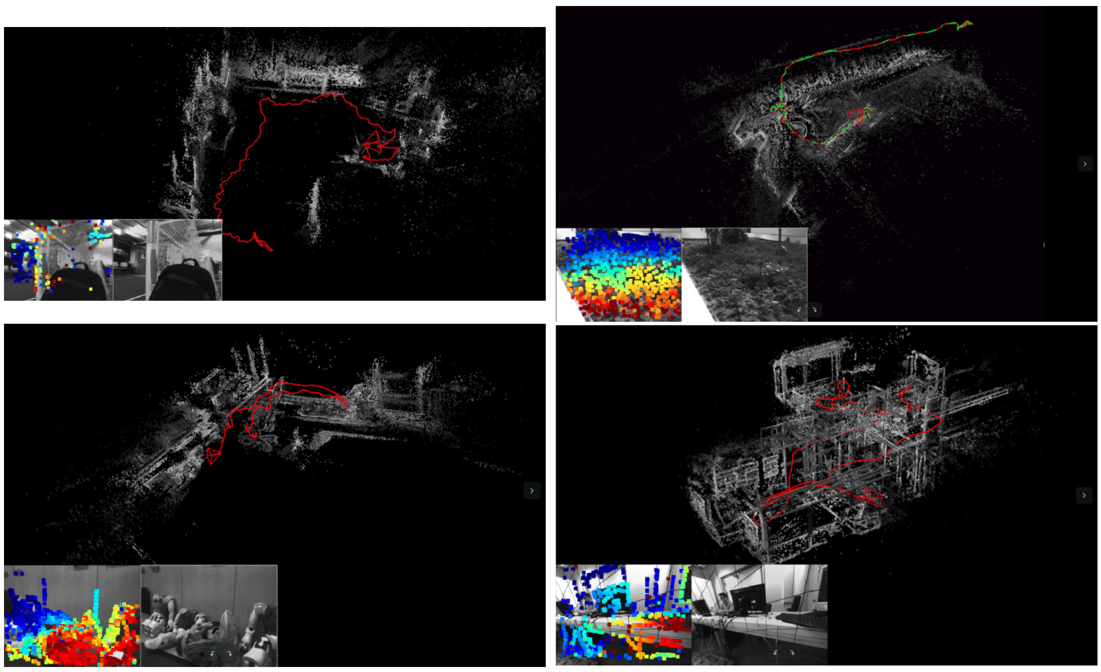


Figure 4.22: Example maps created with DSO method. The top row and left bottom maps come from Nao's real-time video, while the one on the bottom right from a sequence of TUM mono VO Dataset

done in the upcoming frame on the epipolar line for point matching, aiming to minimize the photometric error as seen in Equation (4.28). Every point that is matched is assigned with an initial coarse depth value, which can later change, and an associated variance that describes the satisfaction of the choice's outcome and the need to continue the search to find the current point's match.

3. **Point Activation**. As the direct odometry process evolves, and old keyframes are marginalized, so do the points in them. The story continues with new candidate points to activate, and then being also marginalized. The candidate points that are activated have maximized distance between the already active points, because a nice distribution in the image needs to be maintained. This will continue until the overall system execution is over. It is important to note, that the number of the generated candidate points in each keyframe, and the number of all active points between all active keyframes is the same and equal to 2000, because of the need to have extra points to activate, as we want to keep a balance in the number and

distribution of points through the active keyframes.

In summary, DSO is not a SLAM system, but a visual odometry approach to estimate the camera's motion in space, while at the same time sparsly reconstructing the environment with a joint optimization of all model parameters, including the camera's intrinsics, extrinsics, and points' inverse depth. Pixel intensities decide the creation of points that synthesize the point cloud map and that's why a precise measurement model is needed that is provided through photometric calibrating the camera. Figure 4.22 provides two maps created with DSO. The first three come from the Kouretes Robotics Laboratory[1] and spaces inside the Technical University of Crete, while the right from a sequence of the TUM mono VO Dataset [6].

4.5 Indirect Monocular Visual Odometry

Along with the Direct Sparse Odometry (DSO) system, we have also relied on the indirect visual odometry and open-source system of Mur-Artal and Tardos [3], called ORB-SLAM. This is an extension of their previously produced work in [43], containing the addition of an initialization method, and the Essential Graph, and to today, it is proven to be the most capable feature-based SLAM system.

To be more precise, the latter version ORB-SLAM2 [44] was used in our system, which is an enhancement of the first version ORB-SLAM that was created exclusively for monocular cameras. The ORB-SLAM2 system can support more camera models, such as RGB-D and stereo cameras, but most importantly for us, has improvements over all the building blocks of the system, offering greater computational efficiency and more robust real-time performance.

In a few words, ORB-SLAM system depends on features to do the localization, the mapping, and the loop closing, and these three tasks are executed in parallel on an individual thread. An effective initialization process is used to create the initial map, and a reduction of the whole problem is later done with a covisibility graph, that enables tracking and mapping in a smaller interval than using the whole map. Also, loop closing

[1]http://www.intelligence.tuc.gr/kouretes/web/

is being done with the help of the Essential Graph while real-time relocalization of the camera is supported.

ORB Features

Building on what we discussed in Section 2.4.4 about feature detection and representation, the feature decided to be used in this system is called ORB [25], from Oriented FAST and Rotated BRIEF. This feature was developed in the OpenCV labs and can support real-time point matching while providing good invariance to brightness and camera motion. To enable more efficient results than that of each method individually, for the FAST [45] features, namely Features from Accelerated and Segments Test, an orientation component was added to them. For the BRIEF [46] features, meaning Binary robust independent elementary feature, a method was suggested from decorrelating them under rotational invariance.

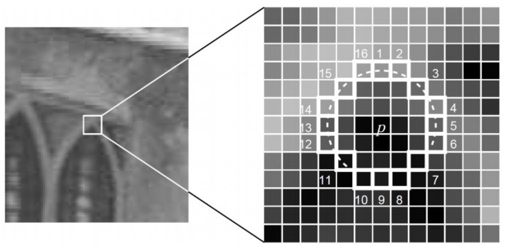


Figure 4.23: FAST keypoint detection process example for a circular radius of 3

In more detail, the FAST points are decided by comparing the brightness value of a pixel p with that of pixels located in a circular area around it. Those pixels are first divided into three cases based on their intensity value in comparison to that of pixel p, and if half of those pixels have darker or brighter value than that of p, then it is selected as a keypoint. A circular radius of 9 was selected because it was seen that it provided better results. In Figure 4.23 we can see an example of this process but with 16 pixels in the circle around point p, as shown in [45].

Because FAST doesn't support multi-scale features, an image pyramid is applied representing the image at different resolutions. For every downsampled version of the image, a detection of keypoints is activated, producing scale invariant points. When a keypoint is located, an initial rotation is assigned to it based on the intensity change around the keypoint described by the *intensity centroid*, which assumes that when the keypoint is a corner, its intensity is offset from the center and thus can reveal its direction. The moments of a patch are defined as,

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y) \tag{4.35}$$

The moments then, can provide us with the centroid, or the center of mass of the patch, as,

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \tag{4.36}$$

Consequently, a vector can be created from the center of the corner to the centroid, giving the orientation of the patch, also shown in Figure 4.24, in rads as,

$$\theta = \operatorname{atan2}(m_{01}, m_{10}) \tag{4.37}$$

The centroid gives a good orientation even with a lot of noise in the image. When θ is computed, a descriptor can be created for the point. Because of the orientation component addition, the detector is now reffered as *oFAST*.

Once the FAST points are extracted, BRIEF will convert them to binary strings or vectors of 256 bits length, all together contained in a binary feature descriptor, to then jointly represent a scene object. A BRIEF descriptor, over a smoothed with a Gaussian distribution around the center patch p, is constructed of binary tests,

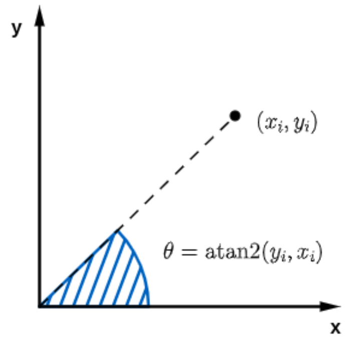


Figure 4.24: Defining the atan2 angle

$$\tau(p; x, y) = \begin{cases} 0 & \text{if } p(x) \geq p(y) \\ 1 & \text{if } p(x) \leq p(y) \end{cases} \tag{4.38}$$

where $p(x)$ is the pixel x intensity in the patch p. Note that before the tests are performed, the whole image is also smoothed by applying an integral image, when each patch is a 31 × 31 pixel window that consists of 5 × 5 pixel test points. We can define the feature then as a vector of n binary tests as,

$$f(n) = \sum_{1<i<n} 2^{i-1} \tau(p; x_i, y_i) \tag{4.39}$$

In order to make BRIEF invariant to rotation, a method to steer it by considering the keypoints orientation is used. Thus, for each binary feature descriptor at (x_i, y_i) of n tests, we need a $2 \times n$ matrix,

$$S = \begin{pmatrix} x_1, \ldots, x_n \\ y_1, \ldots, y_n \end{pmatrix}$$

Then, with patch direction θ computed in Equation (4.37), a steered rotation matrix from S can be defined as,

$$S_\theta = R_\theta S$$

The BRIEF operator is then,

$$g_n(p, \theta) = f_n(p) | (x_i, y_i) \quad \in S_\theta \tag{4.40}$$

Moreover, the angle is discretized to 12 degrees increments, and an offline computation of BRIEF patterns is stored. Depending on the keypoint orientation θ, the best set of points S_θ will be used.

Overall, the algorithm of the rotation aware BRIEF, or *rBRIEF*, which produces better results than that of steered BRIEF in the variance and correlation, is as follows:

- Each test is executed against all selected patces.
- Create a vector T, holding the tests organized based on their ditance from the 0.5 mean value.
- With a greedy search,
  1. The first test in vector T is used in the orientation vector R and then removed from T.

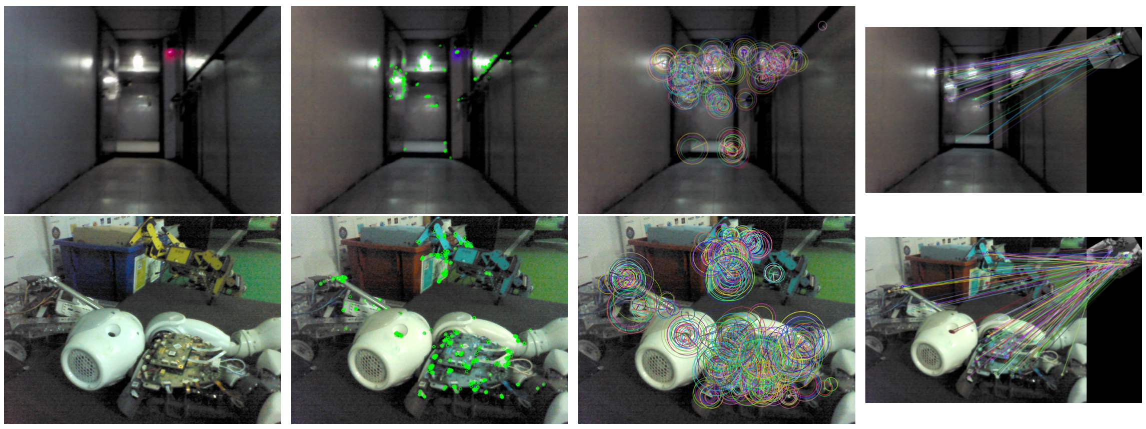


Figure 4.25: ORB feature extraction examples. From left to right, first the initial image, the keypoints with no size, the keypoints with size and color, ORB feature matching with the same image but rotated and blurred

2. Compare the upcoming test in T will all of those in R, and if its absolute correlation is greater than a predefined value then remove it from the set, else put it inside R.
3. We continue executing the previous step until the total number of tests in R is 256.

Overall, ORB is an oriented descriptor, created by using the oFAST detector and rBRIEF descriptor, which is mostly invariant to Gaussian image noise, photometric, and geometric distortions, and is both computationally efficient and fast to match. We can see some examples of ORB feature extraction in Figure 4.25.

It is notable, that for every point p_i, we store its 3D Euclidean coordinates, the ray n_i from the camera optical center to the point, its ORB descriptor D_i that minimizes the Hamming distance in comparison to the other frames the point was viewed, and the minimum d_{min} and maximum d_{max} distance that the point can be measured.

Covisibility Graph

The information between keyframes is maintained with a *Covisibility Graph*, that keeps alive only a sub-set of the total keyframes created, in order to achieve real-time system performance. The graph is operating as an undirected weighted graph, where each active keyframe is represented by a node connected with other nodes though weighted edges. The weight of an edge denotes the common number of observed points in the scene between the two linked keyframes, with minimum value of 15 for an edge to be created.

For loop closing, a double windowed optimization is applied on the covisibility graph, as proposed in [47], that adjusts and distributes the drift error created by the camera motion to the whole part of the graph that links the loop. This optimization framework is done in two windows, first the inner window containing the relation between the camera pose and the points' pose in the scene, modeling the local area, and then an outer window describing the relation between the different camera poses, that aims to balance the camera poses. This method can assist large-scale environment explorations and still produce accurate results that can compare with that of local Bundle Adjustment.

Along with the covisibility graph, ORB-SLAM maintains another graph that is a subgraph of this, called the *Essential Graph*. This graph aims to store only the crucial or necessary information of its ancestor's graph, meaning that all the nodes stay the same but only the edges with weight higher than 100 are being kept. In total, the Essential Graph is built from a spanning tree that is spreading from a starting keyframe, and contains the loop closure links and the just mentioned strong weighted edges. A covisibility graph example is shown in Figure 4.26.

Bag of Words

In addition, a bag of words representation, based on the work of Lopez and Tardos in [48] is used for each feature, to help with efficient camera place recognition, relocalization, and loop closing, by establishing trusted map point correspondences between keyframes.

A visual vocabulary is created offline by extracting ORB descriptors from a big set of images that will be used to represent the real-time extracted points. The bag of words uses this visual vocabulary, that contains discretized from the whole descriptor space visual words, to convert the keyframe into a vector v allowing for greater points in the image manipulation. Chiefly, a database is built by the system progressivelly, forming an *inverse index* and a *direct index*. The inverse index contains for each visual word the keyframes in which it was seen and allows for fast information retrieval about the

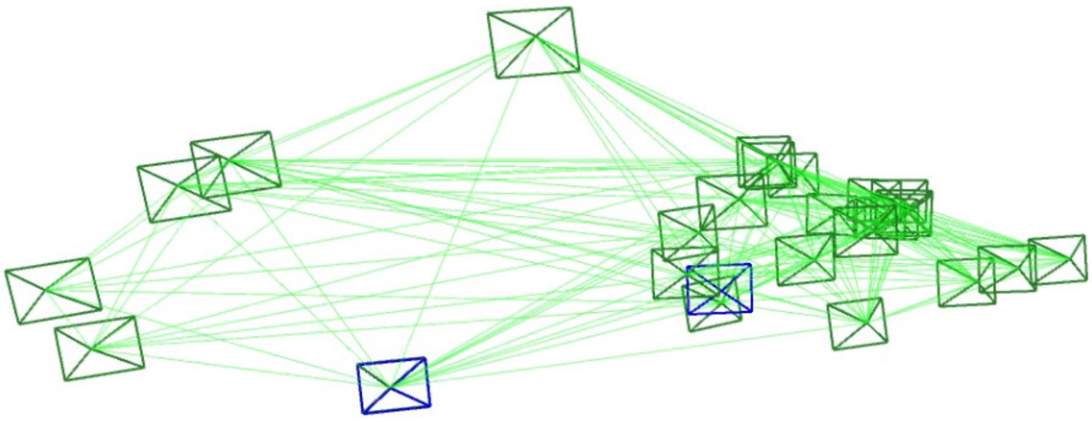


Figure 4.26: A covisibility graph example. The camera poses are linked with covisibility links (green lines)

keyframes, and the direct index provides quick bag of words matching between the initial and the retrieved keyframe.

The descriptors in the visual ORB vocabulary are divided into clusters by using hierarchical k-means clustering, producing the top level nodes of the vocabulary tree, with the number of leaf nodes equal to the number of visual words in the vocabulary. Also, each word is assigned a weight that denotes its rareness in the total collection and is computed by using the term frequency and inverse document frequency, weights that are used to show how important a word is in a corpus. The process of matching keyframes though bag of words is shown in Figure 4.27.

In general, bag of words is a way to group descriptors. If two bag of words are different, then the descriptors don't match. The comparison between two bag of words is fast and thus can help with the loop closing procedure, where a lot of bag of words need to be compared. In fact, when two keyframes have a a lot of common bag of words, they are considered loop closure candidates and will be next examined if they fullfil the necessary consitions for a loop closing to take place, as we will se later on.

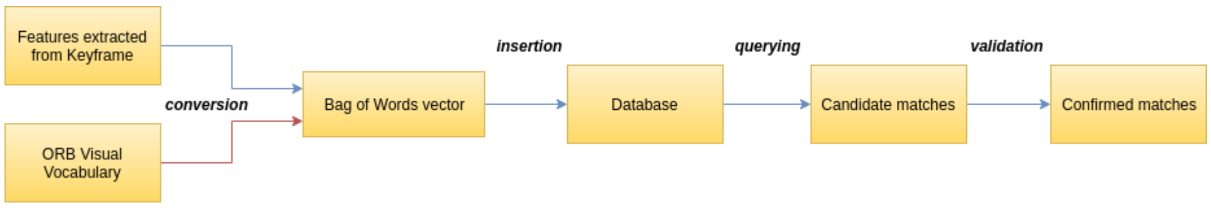


Figure 4.27: Place recognition by comparing bag of words vectors

Map Initialization

The whole system starts with the creation of an initial map, by estimating the camera pose from the first two image frames, and projecting in the image the initial set of observed points with their computed depth from triangulation. Two models are computed at the same time because of the uncertainty of the scene geometry, a homography H, which assumes a planar scene, and a fundamental matrix F, which considers the scene with more complex geometry. From these two methods the needed one will be chosen depending on the scene's nature. We can analyze the process of map initialization as:

1. **Initial point matching**. As discussed in Section 4.5, ORB features are used in the system. The features are extracted from the two initial and consequtive keyframes, and a correspondence of the ones in the first keyframe are searched in the second. In case that not many points match were found, this step is done again.

2. **Models computation**. The homography H, as explained in Equation (2.91), and the fundamental matrix F in (2.93), are computed in parallel. For the homography, the direct linear transformation (DLT) is used with the *four point algorithm*, while for the fundamental matrix, the *eight point algorithm* as discussed in Section 2.5. These methods were operating inside the iterative random sample consensus (RANSAC) scheme. For each model reffered as M (which can be either H or F) at every iteration, a score S_M is computed, as

$$S_M = \sum_i \left(\rho_M\left(d_{cr}^2\left(x_c^i, x_r^i, M\right)\right) + \rho_M\left(d_{rc}^2\left(x_c^i, x_r^i, M\right)\right)\right) \tag{4.41}$$

where,

$$\rho_M(d^2) = \begin{cases} \Gamma - d^2 & \text{if } d^2 < T_M \\ 0 & \text{otherwise} \end{cases} \tag{4.42}$$

Here, the d^2_{cr}, d^2_{rc} denote the symmetric transfer errors from one frame to the next, and T_M is a threshold for outliers rejection, which is a different value for each model.The Γ was selected as the outliers rejection threshold of the homography H model to make the process more fair. In the end, the best score for each model decides that model's representative homography and fundamental matrix.

3. **Model selection**. Because of the many sub-cases that can explain the scene planarity, a heuristic function is computed to decide the end model used as,

$$R_H = \frac{S_H}{S_H + S_F} \tag{4.43}$$

In case $R_H > 0.45$ the homography H will be chosen, that supposes low parallax in points, or in any other case the fundamental matrix F is the suitable model.

4. **Camera pose estimation**. Now that the model is selected, the camera motion needs to be defined between the two keyframes. For the homography case, the eight motion hypotheses is used, by triangulating them and finding the one with the minimum reprojection error of points, considering that the parallax is enough to provide with the appropriate information.

   In case of the fundamental matrix, the essential matrix is computed with the help of the intrinsics camera matrix as,

$$E_{rc} = K^{\top} F_{rc} K \tag{4.44}$$

   Then, the four motion hypotheses are used and by a similar triangulation as done for the homography case, the option with the less reprojection error is chosen.

5. **Full Bundle Adjustment**. Because of the low space of the initial mapping model, a full Bundle Adjustment can be done without any significant delay on the system to improve the reconstruction.

Camera Tracking

As already mentioned, tracking, mapping, and loop closing are operating in individual and parallel threads. In the tracking thread, first an eight level pyramid is created from each image frame with a scale of 1.2, to extract around 1000 FAST corners from our 640×480 image in a uniform distribution. An ORB descriptor is then computed for each FAST corner generated that will be used later.

Assuming that the tracking until the last frame was successful, a velocity motion model is used to approximate the current camera pose and try to find point correspondences from the previous frame to refine it. In case the tracking was lost, the frame is turned into bag of words, and a search is being done in the created database to find an appropriate keyframe candidate for a global camera relocalization. The camera pose can again be corrected depending on the number of inliers.

When a camera pose is approximated, the search for more points in the scene can continue. The point correspondences is maintained inside a local map, which is a subspace of the global map in order to minimize complexity, that consists of the keyframes K_1 that observe a good number of common map points with the current frame, the keyframes K_2 that are neighbors of the keyframes K_1 in the covisibility graph, and the keyframe K_{ref} which belongs to the K_1 set and has the biggest number of common points with the current frame. For every map point in the local map the procedure we follow is,

- The point's projection in the current frame is computed.
- We compute the relation in angles between the ray from the current frame to the point, and the mean ray of the local map. If the angle between them is greater than a predefined threshold we consider the point an outlier and remove it.
- We compute the distance d from the point to the camera optical center, and if that distance doesn't belong in the area of values between the minimum d_{min} and the maximum d_{max} able point measured distance, we remove it.
- The scale of the current frame is then computed as $\frac{d}{d_{min}}$.

- The point tracking is done by comparing the point's computed descriptor with the ORB descriptor that needs to be matched in the neighbor area of that point at the computed scale.

Once this process is done for all points in the local map, with the points observed by the current frame, the camera pose can be estimated with a good confidence.

We must note, that for each keyframe we store the camera rigid body pose from world to camera coordinates, the camera intrinsics matrix, and all of the ORB features that were observed in that frame. The general policy for keyframes, is that they are added at a fast pace so that we will always have spare to provide information. Overall, for a frame to become a keyframe it must complete all of the following requirements:

- At minimum 50 map points can be tracked in the frame.
- In comparison to the keyframe K_{ref}, it must have tracked fewer than 90% of common points.
- At least 20 frames have passed from the last global relocalization and the last keyframe generation.

These requirements can ensure that the tracking procedure will be robust and the computations will stay to the minimum while preserving efficiency. In the end, a motion only Bundle Adjustment is done to optimize the camera pose and minimize the reprojection error as,

$$R, t = \arg\min_{R,t} \sum_{i \in X} \rho \left(\left\| x^i - \pi(RX^i + t) \right\|_{\Sigma}^{2} \right) \tag{4.45}$$

where R and t are the camera's orientation and translation, ρ is the Huber cost function and Σ the keypoint's covariance matrix. It is mentionable that all optimizations applied in this work follow a Levenberg-Marquardt algorithm implementation.

Local Mapping

The local mapping thread is responsible for maintaining a locally consistent map of points and keyframes relation in the covisibility graph. Once a new keyframe is created, it is added into the covisibility graph as a node, and thus updates its relation with the neighboring nodes, and especially, the node with the most common field of view when it comes to points. The keyframe is then converted into bag of words which will later be needed.

In a keyframe, a map point is created by triangulating ORB features though constant camera motion recognized in neighbor keyframes in the covisibility graph, and thus collecting more information about the point's position in the scene which increases our conficence about its computed depth. A match in another keyframe's pixel coordinates is not used if the epipolar constraint is not valid as seen in Equation (2.86).

Also, for each keyframe, not every map point belonging to it will be kept, as only the ones that can really help with the structure from motion process are needed. To decide which point in the keyframe is useful and can provide with trustworthy information about its depth, it must be tracked in at least 25% of the keyframes that it would be logical to be present, and in at least 3 keyframes in general. This way the outliers are removed and errors in localization are reduced.

A Bundle Adjustment is applied in the local mapping model, influencing and refining the measurements of the current keyframe K_i, every neighboring keyframe's of K_i in the covisibility graph, and all the points observed from these keyframes. Long-term computational efficiency is maintained by removing the keyframes that contain, to a significant extent, similar information on points as their close neighboring keyframes as discussed at the end of camera tracking Section 4.5.

Loop Closing

The last of the three threads tries to find in the current frame the possibility that the scene observed is a place that we have already been into, and hence consider closing a loop, while also optimize the course taken by the robot if needed.

First we need to find the keyframe candidates to close a loop. We search the covisibility graph for neighboring to the current keyframe nodes, and for those with matching map points greater than 30, we compute the similarity in their bag of words representation, and keep the lowest score S_{min} produced, as seen in [49]. The score between the two bag of word vectors v_i, v_j is computed as,

$$S(v_i, v_j) = 1 - \frac{1}{2}\left\| \frac{v_i}{|v_i|} - \frac{v_j}{|v_j|} \right\| \tag{4.46}$$

This is not the final score we use, as $S(v_i, v_j)$ will be next normalized with the score computed by a previous keyframe that shares mostly the same view as the initial one,

$$\eta(v_i, v_j) = \frac{S(v_i, v_j)}{S(v_i, v_{i-1})} \tag{4.47}$$

Any bag of words in the database with score lower than that of S_{min} is not taken into account, and as obvious, the keyframes that are close to the current one in the covisibility graph are not considered either. A candidate is selected, only if at least three successive keyframes in the covisibility graph have a score higher than S_{min}.

The similarity transformation describes the accumulated error until the time we close a loop in the 7 Degrees of Freedom (DoF), namely 3 for translation, 3 for rotation, and 1 more for scaling, that a camera can move. The idea is that first the ORB feature correspondences are computed between the current and every candidate keyframe, and then the bag of words of these keyframes are compared for loop closing.

In brief, the group of similarity transformations is the $Sim(3) \subset \mathbb{R}^{4\times4}$ Lie group and is a combination of a rigid transformation, as discussed in Section 2.4.1, and scaling $s \in \mathbb{R}^+$.

If the similarity transformation is found with a good amount of inliers between the current keyframe K_c and the loop closing keyframe K_l, then we optimize it by minimizing the reprojection error e_c, e_l, representing the distance between every point's projected into the image plane pixel and the measured one, of each keyframe as,

$$e_c = x_{c,i} - \pi_c(S_{cl}, X_{l,j}) \tag{4.48}$$

$$e_l = x_{l,j} - \pi_l(S_{cl}^{-1}, X_{c,i}) \tag{4.49}$$

where x is a 2D projected keypoint, X is a 3D map point, and S_{cl} is the similarity transformation between the two keyframes. The following cost function is then to be minimized,

$$C = \sum_n \left(\rho_h\left(e_c^\top \Omega_{c,i}^{-1} e_c\right) + \rho_h\left(e_l^\top \Omega_{l,j}^{-1} e_l\right)\right) \tag{4.50}$$

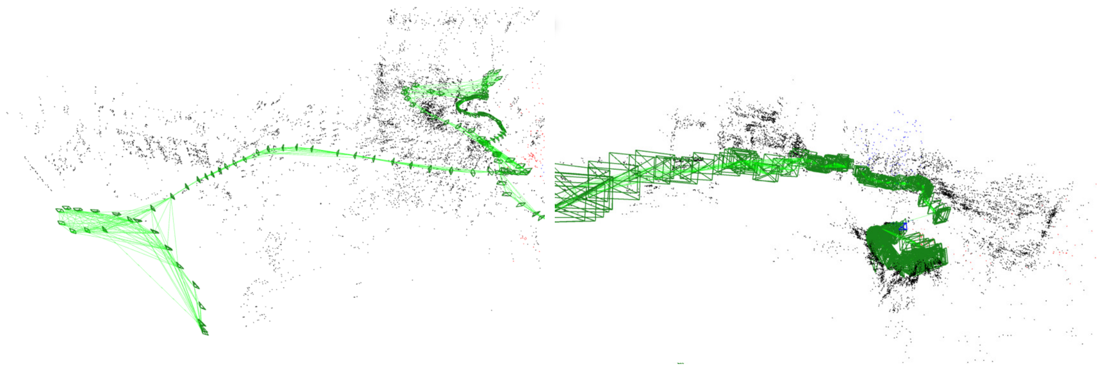


Figure 4.28: Example maps created with ORB-SLAM by using Nao's real-time video with ROS

Here, Ω denotes the covariance matrices connected with the scale the point was observed in each image.

As we assume that the map is static, we expect the points in the keyframe K_l that closes the loop to be for the most part the same. The covisibility graph is updated to recognize this keyframe not as a new one, but as an already existing one K_c, and thus the node stays the same, but the edges are adapted to obey this event. Next a chain reaction of refinement in the keyframe and its neighbors is done, based on the similarity transformation. At last, the loop is corrected with a pose graph optimization on the Essential graph improving both the point correspondences and the camera pose at each keyframe.

After a loop closure, a full Bundle Adjustment is applied to optimize all the keyframes and points of the mapped environment. In Figure 4.28 we present two maps that were created from the Nao robot's real-time video by using ORB-SLAM in spaces of the Technical University of Crete. With black we represent the already mapped features, while with red the currently processed ones.

Overall, ORB-SLAM is a full simultaneous localization and mapping system that can do tracking, mapping, and loop closing. Even if as a system, it is proven to be very efficient and accurate through experiments, especially for indoor environments, and the ORB features are highly robust to photometric variations and scaling, when the monocular camera motion has a lot of rotations and there is little texture to the scene, it can still fail the camera pose estimation. Like with the DSO system, all computations are also happening in parallel in the CPU.

4.6 Coupled Semi-Direct Monocular SLAM

Our total SLAM system for Nao robot consists of both the DSO system, as presented in Section 4.4, and the ORB-SLAM system, in Section 4.5, which are coupled as proposed by Hun Lee and Civera in [1]. A graphical representation of the total system is also provided in Figure 4.29.

Because direct and indirect visual odometry methods have their own limitations and benefits, we can create a combination of both to produce a better performing one. As discussed by a recent work in [50], we can evaluate their performance on the basic aspects of photometric calibration, motion bias, and rolling shutter effect,

- The *photometric calibration*, which we have already seen in Section 4.2.2, can be done to a camera to correct a projected 2D in the image point's intensity value which may be distorted due to vignetting, exposure or gain. The direct methods seem to remarkably improve their performance when a photometric calibration is available, whereas the indirect ones are not always influenced, because it is based mostly on the type of the features and the way they are extracted. However, knowledge over the camera response function can still positively affect them.

- The *motion bias*, refers mostly on the way a visual odometry method behaves when a dataset is examined not only in a forward way, but also backwards. In contrast with the direct methods, the indirect have a big decrease in performance when it comes to this issue. A reason to this for example is image artifacts. Therefore, higher resolution images are needed.

- The *rolling shutter effect*, which is caused due to motion in the scene when a video frame, which by a electronic device is scanned either vertically or horizontally, hasn't completed on taking a snapshot. Thus, distortions will be introduced in the image which a geometric or photometric calibration will not be able to correct. Both the visual odometry methods are sensitive to this effect, but the one that suffers the most is the direct one. Indirect methos tend to be more robust in the presence of rolling shutter.

To compensate for these aspects, DSO and ORB-SLAM are combined in a semi-direct visual odometry system in which they share their advantages, while having greater performance in comparison to when they operated as individual systems. In brief, we maintain

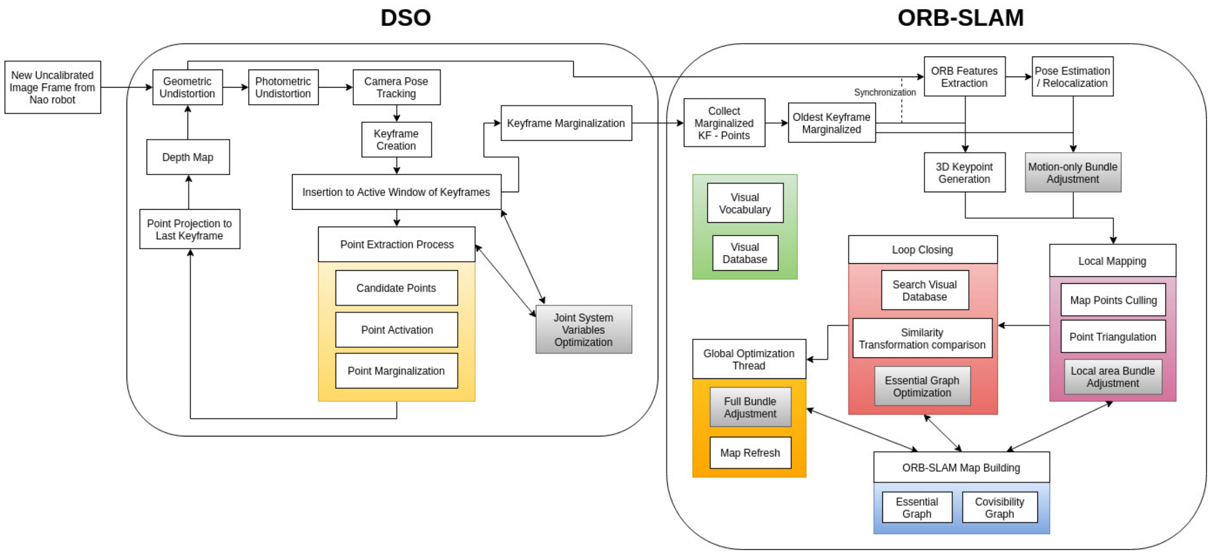


Figure 4.29: Coupled DSO and ORB-SLAM system overview

in real-time two seperate maps, a semi-dense for DSO and a sparse for ORB-SLAM, which contribute online depending on the information needed. For the most part, DSO is used for camera tracking in a local active keyframe neighborhood, as it has great precision and small motion drift, and ORB-SLAM is used for jointly refining a larger scale of keyframe poses, while also providing relocalization and loop closure capabilities.

Both ORB-SLAM and DSO are the best performing methods in their respective fields until today, when it comes to published works. The combined optimization framework of the total system can be broken down into 3 basic layers,

- The *Local level*, for which we adopt the joint optimization of all model parameters of DSO, including the camera pose, intrinsic parameters and observed points' in the keyframe inverse depth, that is done in a sliding window where all old camera extrinsics and points that share no information with the current keyframe are marginalized. New keyframe extraction happens as discussed in Section 4.4.
- The *Mid level*, where after the camera pose and the points in the image are marginalized from DSO, they are sent to ORB-SLAM. Then, ORB features are extracted from the image frame and a motion only optimization on the camera pose that

aims to minimize the reprojection error, as discussed in Section 4.5, takes place. A number of these keyframes with their respective points are added to the map and a local Bundle Adjustment is then applied to optimize the covisible keyframes with their points.

- The *Global level*, which is the last optimization layer, and does first a pose graph, or Essential graph, optimization over the similarity transformations correcting the scale drift created specifically by the monocular SLAM, and then a full Bundle Adjustment as discussed in Section 4.5.

Real-Time ROS Activation

The process begins with the real-time images from Nao's top camera, which have VGA resolution of 640×480 at 30 frames per second, being transferred to our computer station through the ROS framework, as discussed in Section 4.1. The *ROS Master* is executed on our laptop, to which is provided our local computer's IP and Nao's IP with a predefined open $port = 9559$. Thus all computations are done on the computer system's central processing unit CPU, and are being maintained there by administering all the needed running tasks inside a multi-threading architecture for system efficiency. The graphics processing unit GPU, even though more powerful and productive with its highly parallel structure compared to the CPU, did not contribute to the system's computations acceleration.

The *naoqi-driver* ROS package [36] links the robot with the computer, in which the ROS middleware operates, continuously transporting, including others, $sensor\_msgs/Image$ messages, defined in Table 2.1, on the $/image\_raw$ topic which the software receives.

Before starting operating the total system, we provide it with the appropriate files based on the nao camera, coming from the geometric and photometric calibrations done in Section 4.2. In more detail we use,

- A *camera.txt* file, that contains our camera's focal lengths (f_x, f_y) and the optical center (o_x, o_y) produced by a geometric calibration. The input and output image width and height are also defined, with the mode to specify the rectification mode, further discussed in Section 4.2.2. For our system we kept the same input and output image resolution, thus no need to define a rectification mode existed.

- A *pcalib.txt* file, that contains the inverse camera response function U in a single row of 256 increasing distribution of values, further discussed in Section 4.2.2 and as shown in Figure 4.7.

- A *vignette.png* image file, that contains a monochrome 16-bit image describing the smoothed and calibrated vignette function as pixelwise attenuation factors, further discussed in Section 4.2.2 and as shown in Figure 4.11b. Both this image and the *pcalib.txt* file are produced by a photometric calibration.

In total, all the information needed from the subsystems to operate are being pulled and collected by specialized files for each, that will activate their execution. Namely, a ROS *launch* file synchronizes the information that will activate DSO subsystem, while a *YAML* file is used for ORB-SLAM. These filter applications will happen after the images are turned to grayscale for quicker processing.

Direct Subsystem

As the images are in real-time transported through the network with ROS, they will next be welcomed from the DSO system. In general, DSO is assisting the coupled system by being responsible for map initialization, and providing with camera poses when needed to reinforce tracking, and with already marginalized points when the local area mapping requires it. Because we maintain both maps for DSO and ORB-SLAM, anything that DSO contributes with to the end SLAM is basically taken from the map it creates online.

The incoming images are cloned and separated at each frame based on the system they will be used from. Namely, a geometric and photometric correction is applied on them in the case of DSO, else they will be sent directly to ORB-SLAM, and a geometric correction will only be applied on them there. The images are also discriminated and filtered by their *timestamp* value, which denotes the time the image was received by the system. This is important, as for various reasons an image with older timestamp than the one we currently process can be inserted in the system, and if by mistake is handled at that time, it can cause serious damage to the visual odometry procedure.

DSO builds its map in a semi-dense fashion. There is no real advantage on inserting more data to the system to create a denser map of the environment, as it was shown that increasing computations because of the addition of data may indeed make the map denser, but it will not make the camera tracking accuracy more efficient, and thus can be more costly than beneficial.

From what we discussed in Section 4.4 about the DSO system architecture, the photometric error of a point p observed in two frames, is computed as the gaussian weighted sum of squared differences of an eight pixel neighborhood N_p that describes that point, and is given by the Equation (4.28). The projected point's p' position, can still be represented by the Equation (4.29), that depends on the intrinsic camera parameters and the camera motion created between the two frames.

Because in our case a photometric calibration is available, a prior aiming to make the affine brightness parameters close to zero can be added to the total cost function that needs to be minimized. We can describe that function, with the addition of the total photometric error between all keyframes and points as,

$$E_{photometric} = \sum_{i \in F} \sum_{p \in P_i} \sum_{j \in obs(p)} E_{pj} + \sum_{i \in F} (\lambda_a a_i^2 + \lambda_b b_i^2) \tag{4.51}$$

The system variables optimization is then done with the iterative Gauss-Newton algorithm, with an update function as shown in Equation (4.32). Overall, the keyframes initialization, extraction, and marginalization, as well as the point candidate selection, tracking, and activation are as presented in DSO overview of Section 4.4. Additionally, a point is marginalized when it is not observed in the next two keyframes from the last time it was seen.

We must note, that the information extracted from DSO is first converted into ROS messages with a wrapper, and then being communicated to ORB-SLAM through topics inside the ROS framework. That information has to do with the marginalized keyframe parameters, from which, the most important are the keyframe's timestamp, pose, and the points' in the keyframe Euclidean position, and their inverse depth estimation as presented in Section 4.3.

In the worst case that camera tracking is lost, DSO resets and tries to initialize the map from the start.

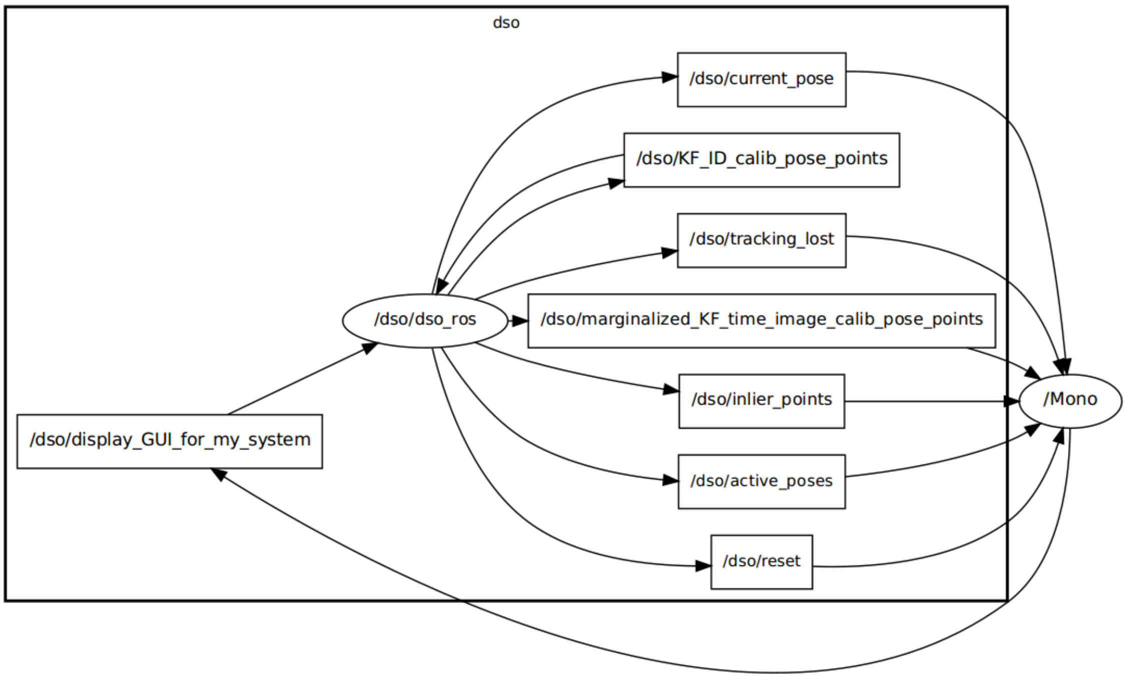

dso
/dso/current_pose
/dso/KF_ID_calib_pose_points
/dso/tracking_lost
/dso/dso_ros
/dso/marginalized_KF_time_image_calib_pose_points
/dso/inlier_points
/Mono
/dso/display_GUI_for_my_system
/dso/active_poses
/dso/reset

Figure 4.30: Internal DSO and ORB-SLAM systems communication supported by the ROS framework. The ellipsoids represent each system's main node, while the rectangles are the topics that transfer the information to these nodes

Indirect Subsystem and Total Coupled System

Even before the DSO system is executed and starts receiving the Nao images, the ORB-SLAM has began operating to load the visual vocabulary of ORB descriptors. At the time it is loaded, it waits until DSO is ready to transfer the marginalized keyframe data and thus the coupled operation to begin. In general, ORB-SLAM contributes to the whole procedure by complementing what DSO offers, meaning global level camera pose consistency, loop closing, feature mapping, and relocalization. The end map is also being completed inside the ORB-SLAM system, and we will discuss its creation here. A graphical view of the inner DSO and ORB-SLAM communication is provided by using the *rqt_graph* ROS tool in Figure 4.30. As shown, the DSO main node *dso_ros* provides the ORB-SLAM main node *Mono* with the data we discussed in the previous Section.

Because in the runtime we keep alive both the DSO and ORB-SLAM map, different scale estimations are computed from both based on the same scene due to the nature of the monocular vision. However, our final map can have only a single estimation and thus we use a scale factor s that denotes the relation between the scale estimates of the two systems. This scale now is used for the final map's similarity transformation $Sim(3)$ estimation between two keyframes, also discussed in Section 4.5, by being applied on the DSO predicted camera motion, and then sent to the ORB-SLAM in an initial similarity matrix as,

$$S_F = \begin{pmatrix} \mathbf{s}\, R_D & t_D \\ 0_{1\times 3} & 1 \end{pmatrix} \tag{4.52}$$

where D and F denote the Direct and Feature-based visual odometry systems. Continuing, because DSO is taking over to the total system's initialization, the map points produced will on that moment be needed and applied on the end map. Points from DSO will also in real-time be inserted into the end map if needed, to create a more smooth distribution of points over the mapped scene together with the features selected from ORB-SLAM. Also, because ORB features can not be extracted from low textured areas, but points must be mapped for consistency, DSO points will be added to dense up the map.

Overall, the coupled decision over which points will be used for the end map, is done by selecting each ORB feature point p from the ORB-SLAM map, assigning it with an inverse depth value that comes from DSO based on that point, and then projecting it to the end map as,

$$x_w = S^{-1}\Pi_c^{-1}\left(p, \frac{d_p}{s}\right), \quad d_p = \frac{\sum_{k\in P_p} \frac{d_k}{\sigma_k^2}}{\sum_k \frac{1}{\sigma_k^2}} \tag{4.53}$$

Here, d_k is the inverse depth of a map point in DSO, that has the same projection position to a point p in ORB-SLAM. A dynamic allocation of points is done to every keyframe for the total system, that decides the number of points mapped in each, based on the frequency the keyframes are added. Namely, if keyframes are added in a quick manner, we use the least amount of points that describe a keyframe, which is 1500, while if they are added slowly, we use the biggest amount of points, which is 2500. Depending

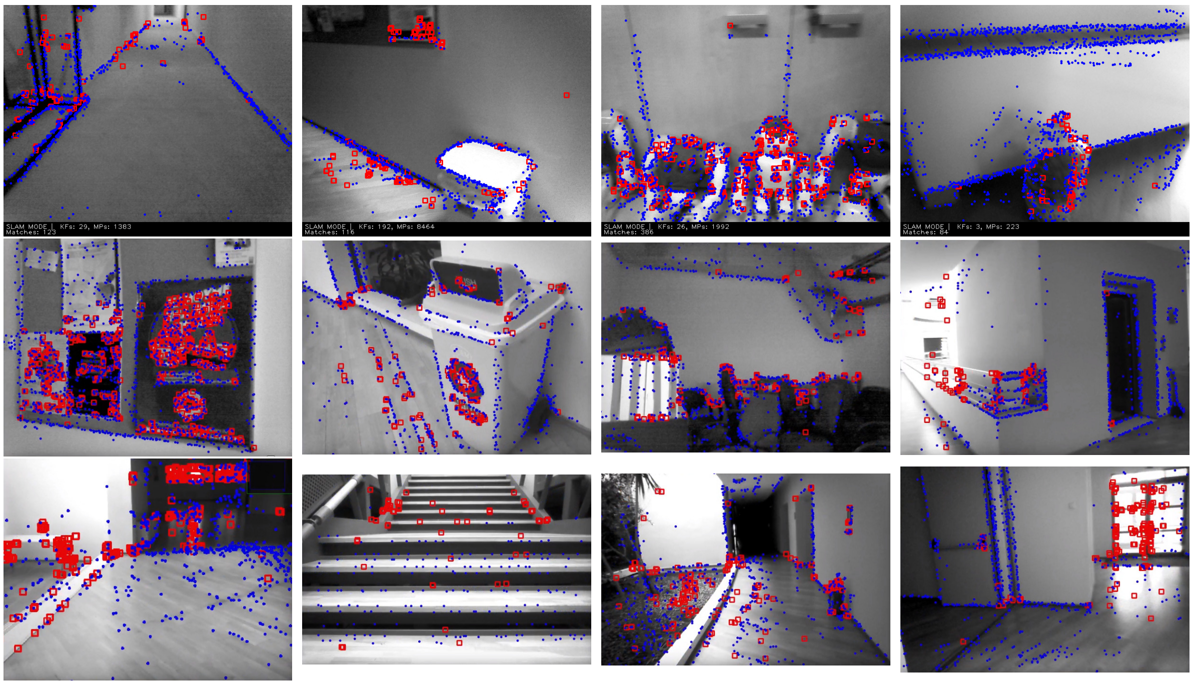


Figure 4.31: Coupled system's point extraction from the real-time Nao video sequence. Both ORB-SLAM features (red) and DSO map points (blue) are used in the SLAM process

on the frequency of keyframes addition, the appropriate number of points is used between those values to represent them. The added keypoints to the ORB-SLAM first system, are also being manipulated by the bag of words ideology presented in Section 4.5. Figure 4.31 provides some examples on the coupled system's point extraction taken by the real-time Nao robot's SLAM process. The red squares refer to the ORB-SLAM features, while the blue dots to the DSO map points.

The policy followed for keyframes starts with the initialization from DSO that provides, among others, the marginalized camera pose estimation. By considering the local area of keyframes in ORB-SLAM, we apply a Bundle Adjustment on the camera motion and intrinsics to refine the pose with the points observed. Therefore, a reprojection error needs to be minimized, between two keyframes i, j, with the following cost function,

$$E_{reprojection} = \sum_{i \in F_{local}} \sum_{x \in P_i} \sum_{j \in obs(x)} \left\| \frac{p_{j,x} - \Pi_c(S x_w)}{\sigma_x^2} \right\|_\gamma \tag{4.54}$$

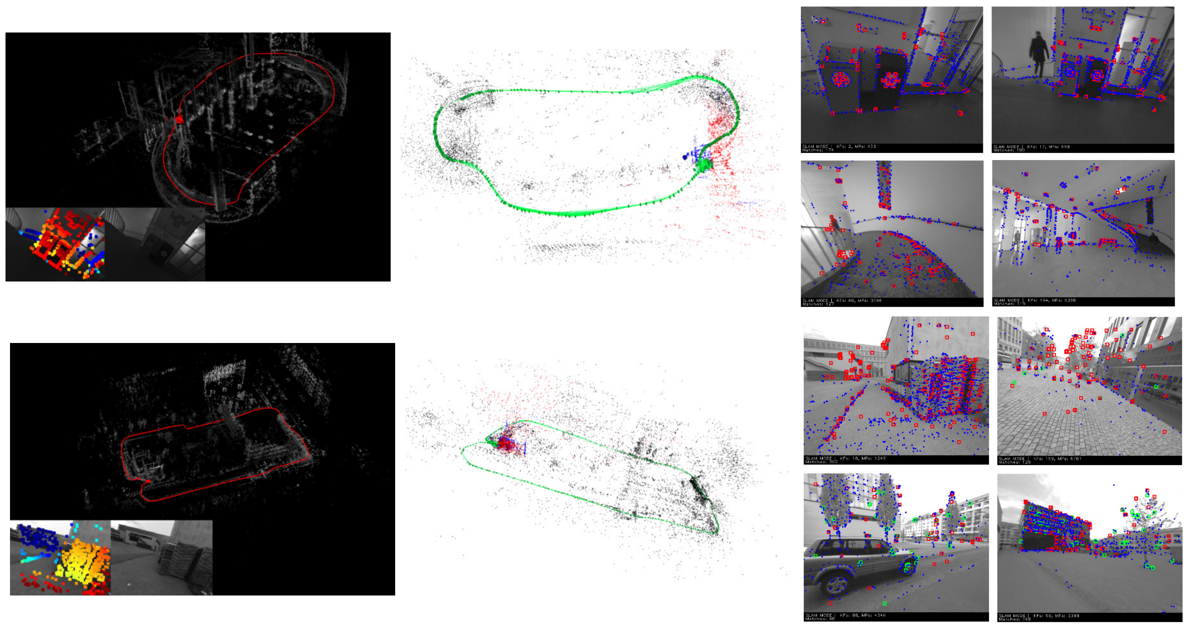


Figure 4.32: Two examples of map generation for the coupled system. From left to right, the DSO map, next to it the final system's combined map, and on the left, images with points extracted from the sequence

As shown, we also consider the variance σ of a keypoint x observed in frame i in the estimation. The variance σ_x^2 of a keypoint can be given by $\lambda_{pyrm}^{2L_{pyrm}}$, thus depends on the pyramid scale factor λ_{pyrm} and the level L_{pyrm} of the pyramid that the keypoint x was extracted from.

The keypoints of Equation (4.53) and the refined keyframes we just talked about will only be used in the end and coupled map that is an extension of the ORB-SLAM map, if the total point correspondences for the last three keyframes has fallen lower than 150. Because of the unique way each subsystem extracts its points from the scene, we can expect that we will be able to provide the end map with a standard number of keypoints, considering that each subsystem can cover for its partner's disadvantages, for environments of many different nature.

The loop closing process of ORB-SLAM is also being exploited by the coupled system, that uses the bag of words for keyframes representation to detect a loop and then with a Bundle Adjustment over the local area of keyframes optimize it, that basically aims to minimize the cost function of Equation (4.54).

The scale drift created by the monocular visual processing is being refined with a pose graph optimization, or Essential Graph optimization, over the similarity transformations as,

$$E_{EssGraph} = \sum_{i=SGedge, j=EGedge} \left\| log_{Sim(3)}(S_0 S_{jw} S_{iw}^{-1}) \right\|_2^2 \tag{4.55}$$

where the $SGedge, EGedge$ indicate the starting and ending edge in the Essential Graph that each frame i, j represent. The S_0 is the global reference similarity transformation, the only one with scale equal to 1, based on which the graph optimization is applied. In the end, a full Bundle Adjustment is done as in 4.5.

Like with DSO in Section 4.6, if in the worst case the camera tracking fails, usually due to low textured scenes, the initialization of the subsystem will need to be done again until it tracks the current camera pose by using the marginalized data received from DSO. We show two examples of the coupled system in Figure 4.32 by using the TUM mono VO Dataset [6], where each row represents one sequence. In the combined map (middle image of each row), the extracted in the map ORB-SLAM features are shown with red, while the DSO map points with blue. Also, the green lines describe the covisibility graph that link the keyframes we choose.

Map loops often exist in big environments. In case of small exploration worlds, loop corrections don't happen, so do the graph or global level optimizations, and thus the camera motion drift and point scale drift increase. To prevent the accumulation of those errors, the keyframe trajectory of both systems is available to provide with the needed pose information if necessary. Between the two trajectories, if loops are detected and the local bundle adjustment is applied, we expect the trajectory of ORB-SLAM to be more efficient than that of DSO. In Figure 4.33 we provide two additional examples of the coupled system by also using the TUM mono VO Dataset [6], where a loop has been detected and then closed. Each of the two sequences occupies two rows.

To summarize, ORB-SLAM is a full simultaneous localization and mapping system that can do tracking, mapping, and loop closing, in contrast with DSO, which is a direct visual odometry system that can not close loops or do relocalization, but can efficiently map points in inverse depth and do robust camera tracking. The coupled system described

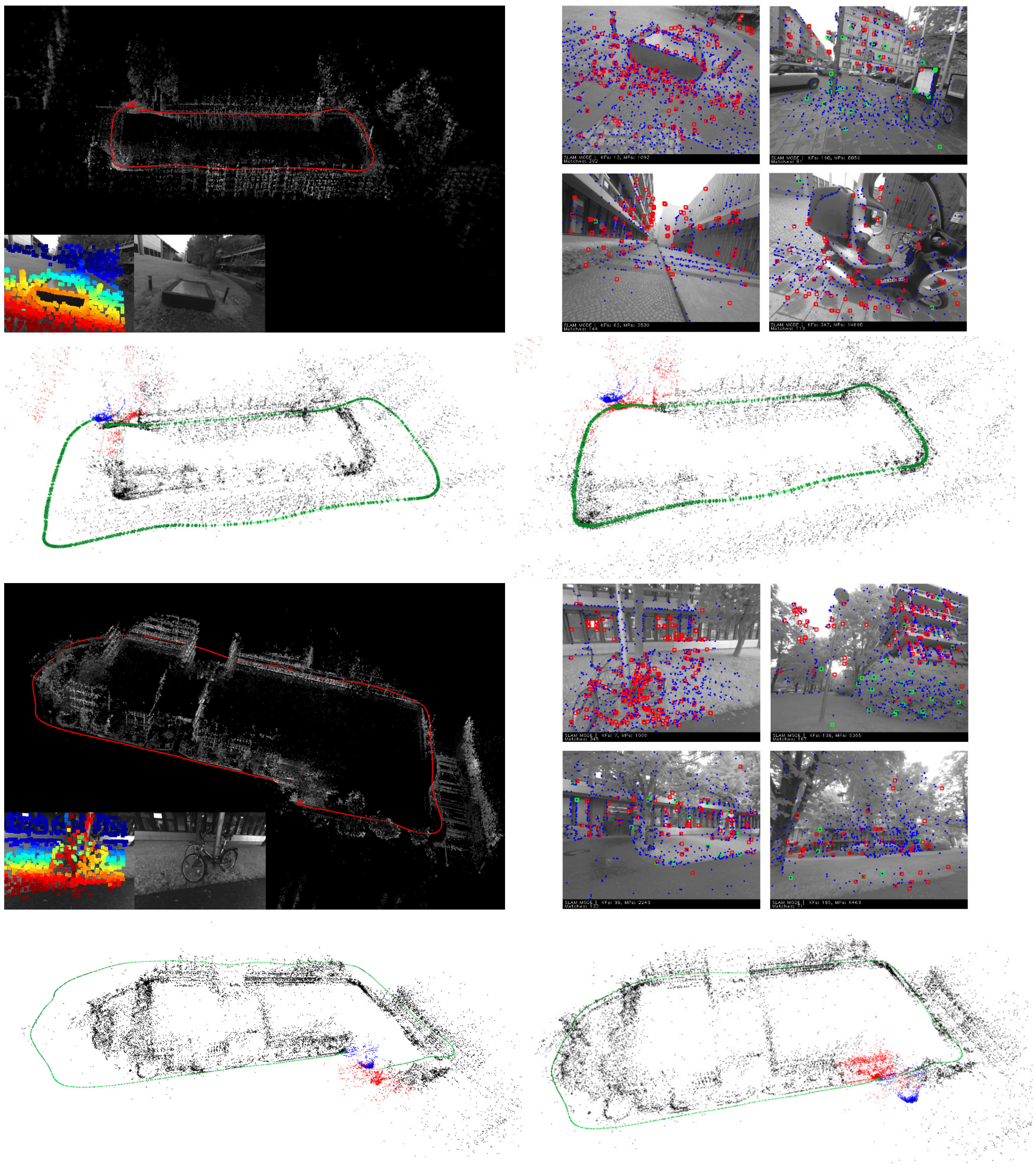


Figure 4.33: Loop closing from the coupled system. The second row of each sequence shows on the left when the map has detected a loop, and on the right when the loop connection is applied

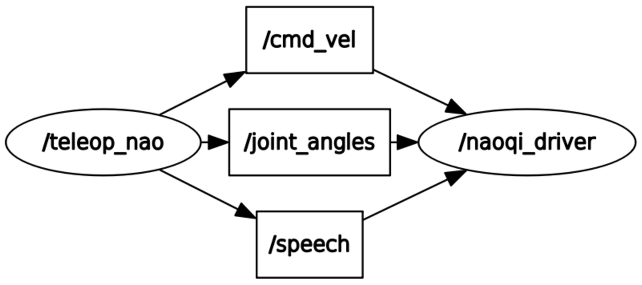
here combines all these attributes in a more performative architecture that is applied to our Nao humanoid robot, aiming to solve the SLAM problem in the full constraint, and cost efficient, case of only a single camera being available.

4.7 Humanoid Robot Teleoperation

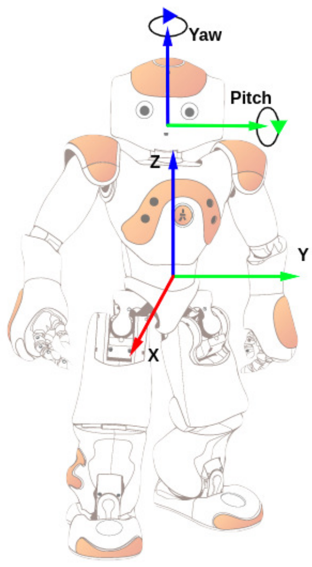
In order to simulate an autonomous agent's navigation, a teleoperation package was created that operates inside the ROS framework as shown in Figure 4.34a. This node is built to run in real-time and parallel to the whole coupled SLAM system, taking over the Nao robot's movements through the computer station's keyboard. Briefly, the *termios* $C++$ header was used to achieve that, which is a Unix API used by the terminal I/O interfaces. Commands given by the keyboard can then asynchronously communicate with the robot to make it move at will.

In total, we take control over both the 2 DoF of Nao's head, which are the pitch and yaw rotation, and over its whole lower body that allows for 2 dimensional robot motion as represented in Figure 2.4. In Table 4.3, we can see the two different types of messages that are used to communicate with the robot's limbs.

Firstly, the teleoperation node sends velocity commands though *geometry_msgs/Twist*



(a) Teleoperation node's communication graph inside the ROS framework



(b) Nao robot 3D pose with head rotation angles messages to the */cmd_vel* topic, that reach the robot's torso and control its body's motion direction. Commands to walk forward, left, right, while also sole clockwise and anti-clockwise rotations are supported, with a spare command to stop the robot's current action in case something goes wrong. A velocity command is broken down into two parts, the linear element, which is described by the Euclidean three parameter XYZ, and the angular element, also described by the Euclidean three parameter XYZ rotation. To activate the 2 dimensional robot motion, for the linear part we move only on the x and *y-axis*, while for the angular part, we only use the z component.

The default speed used for these motions, is 30° a second for the angular motion, and 5cm a second for the linear motion, which can further be changed in the package's *.launch* file as provided for user operation. As seen, the linear movement of Nao robot isn't very fast, first because of its weight, which is 5 kilograms, and second because of the difficulty to stabilize itself in two legs during that motion.

A geometry_msgs/Twist ROS Message	
geometry_msgs/Vector3	*linear*
geometry_msgs/Vector3	*angular*
A /JointAnglesWithSpeed ROS Message	
std_msgs/Header	*header*
string[ ]	*joint_names*
float32[ ]	*joint_angles*
float32	*speed*
uint8	*relative*

Table 4.3: ROS geometric messages needed to control Nao robot's head and body motion.

When the node wants to communicate with the robot to change the camera's field of view, it sends *naoqi_bridge_msgs/JointAnglesWithSpeed* messages to the */joint_angles* topic. To decide what specific joint we want to control, from the total number of 82 referred coordinate frames available, we must first declare it to every message sent. The total coordinate frame tree of Nao robot is shown in Figure 4.35. As already mentioned, the joints we are interested in are the ones managing the head's pitch and yaw rotation. In every message we also specify the max velocity that the head will be able to move, and if the difference in angle between transmitted messages is relative or static. For the messages to be time recognizable from the ROS framework, we set each message's timestamp to the time it was created.

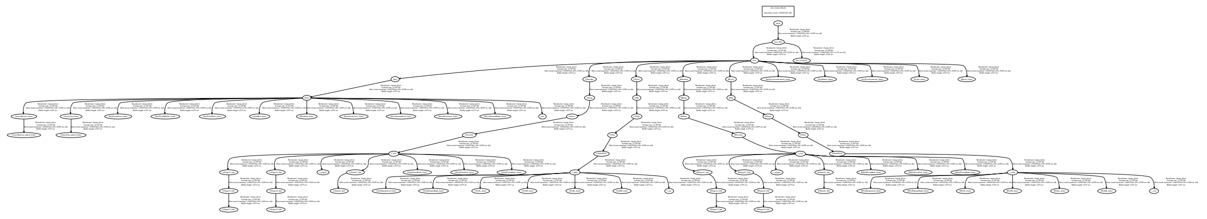


Figure 4.35: Total coordinate tree for the humanoid Nao robot

For the head rotation in the 2 degrees of freedom, we use a default value of 0.1° a second, which like with the body motion, can be adjusted in the package's *.launch* file. We must note that there are limits in the head rotation over the two axis. For pitch, the rotation can fluctuate from −38.5°, front motion, to 29.5°, while for the yaw, the rotation fluctuates from −119.5°, right direction, to 119.5°.

We want the head's rotation speed, created by a single message, to be pretty low, so that a lot of them have to be combined to create a significant change in the head's 3D pose. This way, we support smooth head motions, without fearing that the visual SLAM system will encounter image discontinuities, later lead to camera tracking lost, new map initializations, and overall reduction of system's efficiency and performance.

Chapter 5

Results

In this chapter we present the results of our approach on some indoor and outdoor environments explored by our Nao humanoid robot. The spaces we map come from the Kouretes Robotics Lab[1] and the facilities of the Technical University of Crete. We have included environments with different texture, brightness variation and depth complexity of objects in the scene, to showcase the method's robustness and real-time performance. The standard Figures provided in this Chapter will have on the top left the explored environment's DSO map, on the top right the coupled method's map, and on the bottom row some images during the point extraction process.

5.1 Indoor Environments

We started by testing the final coupled system in restricted spaces and by making small camera motions. For cases like these, the scale s that was first introduced in Equation (4.52), stays close to its initial value 1 and the camera can effectively track its pose during the robot motion, as the trajectory isn't too big.

Scenario no1: Nao robot keeps observing a scene from different viewpoints and tries to map it. Figure 5.1 shows some of the Kourete's Lab robots that are static in the environment. The resulted scene reconstruction of our coupled method is more dense than usual, as we are continuously adding keyframes and all the points assigned to each

[1] http://www.intelligence.tuc.gr/kouretes/web/

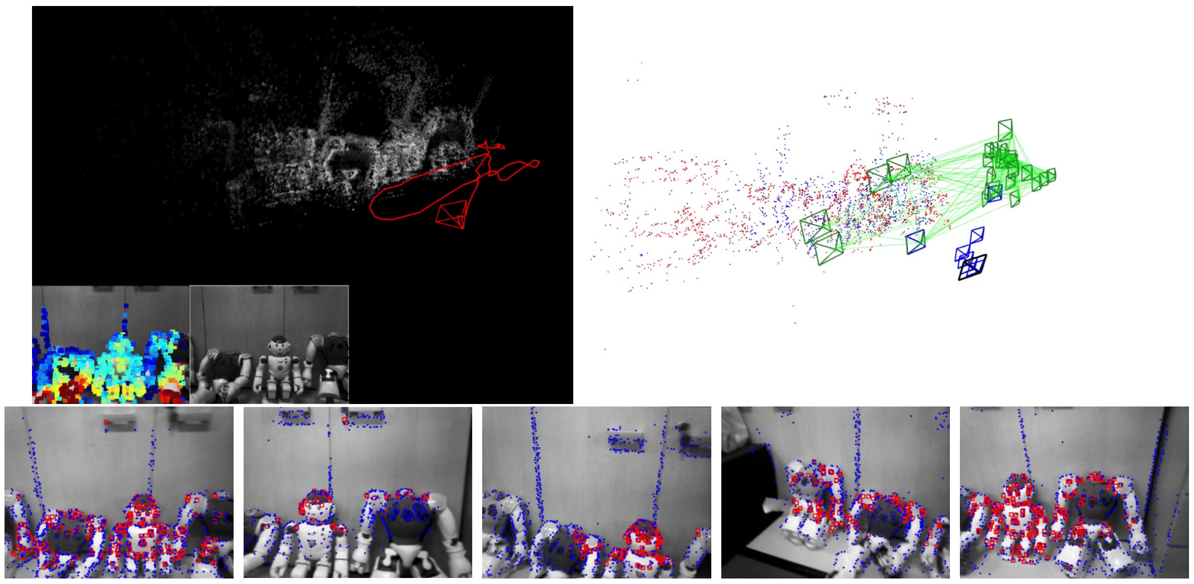


Figure 5.1: Small indoor reconstruction of Kouretes Lab robots

keyframe refer to the same scene. This way, the depth of the observed points is approximated in a greater degree and that is why we can see texture in resulting the map even though our method is a sparse one.

Scenario no2: Nao robot tries to track its camera's pose during a motion that includes a corner in the Electrical and Computer Engineering (ECE) department's building. This is a motion that can cause the camera tracking module to fail, as the robot does a direct 90° turn with the extracted points in each frame changing quickly. Therefore, in order for our robot to not lose its location estimation, quick and efficient point matching must be done. As we can see in Figure 5.2, the camera tracking has succeded and the environment has been sparsly mapped. In the DSO map (top left), we can see in green the full camera trajectory on top of the red trackable camera trajectory, for which some of the keyframes are marginalized.

As we move from smaller to bigger in size environments, the scale s starts to deviate from its initial value depending on the camera tracking and the points matching during the robot motion. The tracking procedure is also getting more difficult as the errors accumulate during the exploration and can thus create a completely wrong representation of

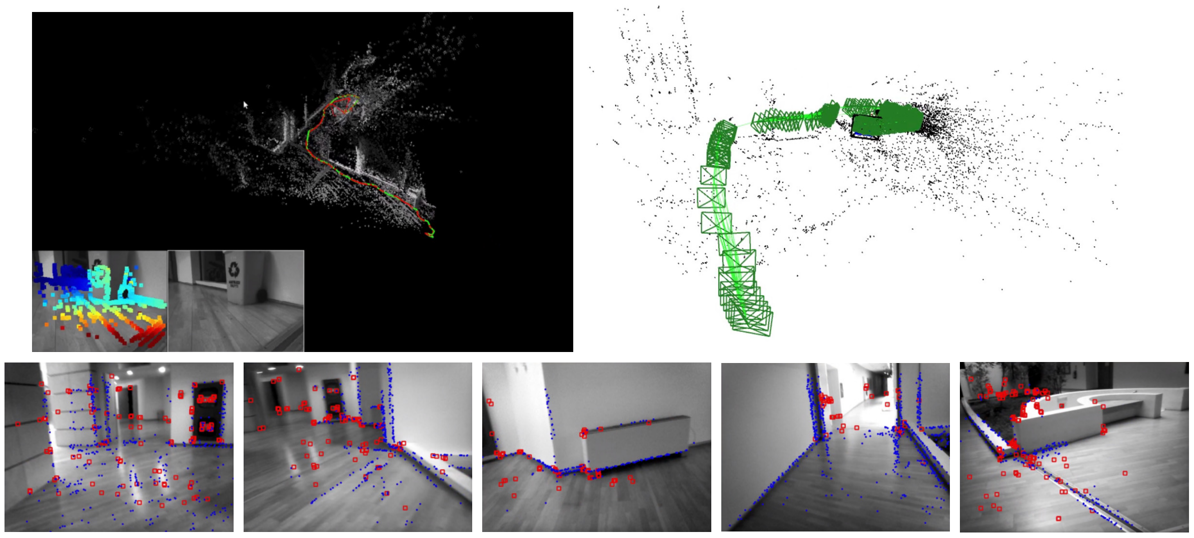


Figure 5.2: Small indoor environment reconstruction through a 90 degrees turn at the ECE department

the environment. However, our proposed system can still manage to operate under these circumstances. We will now provide some results on bigger scale indoor environment mapping.

Scenario no3: The robot will complete a cyclic course that contains a great brightness variation in the middle of it, but will still keep tracking the camera and continue exploring with minimum drift as shown in Figure 5.3. The full camera trajectory for the DSO map is also provided. For this example, three 90° turns have been made to test our method.

Scenario no4: This time, the environment's brightness around the robot will be mostly constant. This space represents the Kouretes Robotics Lab and Figure 5.4 shows the attempt of Nao trying to map it. There is a big complexity in object depths in the background and the camera trajectory is not very smooth, but we are still able to get for the most part accurate results.

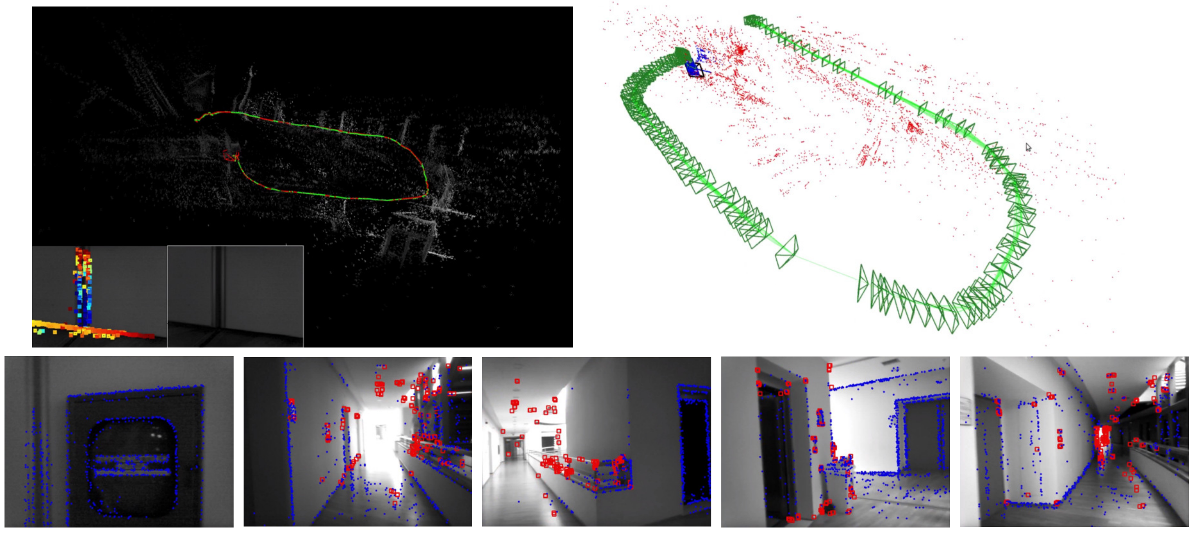


Figure 5.3: Big indoor environment reconstruction by making three turns of 90 degrees at the ECE department

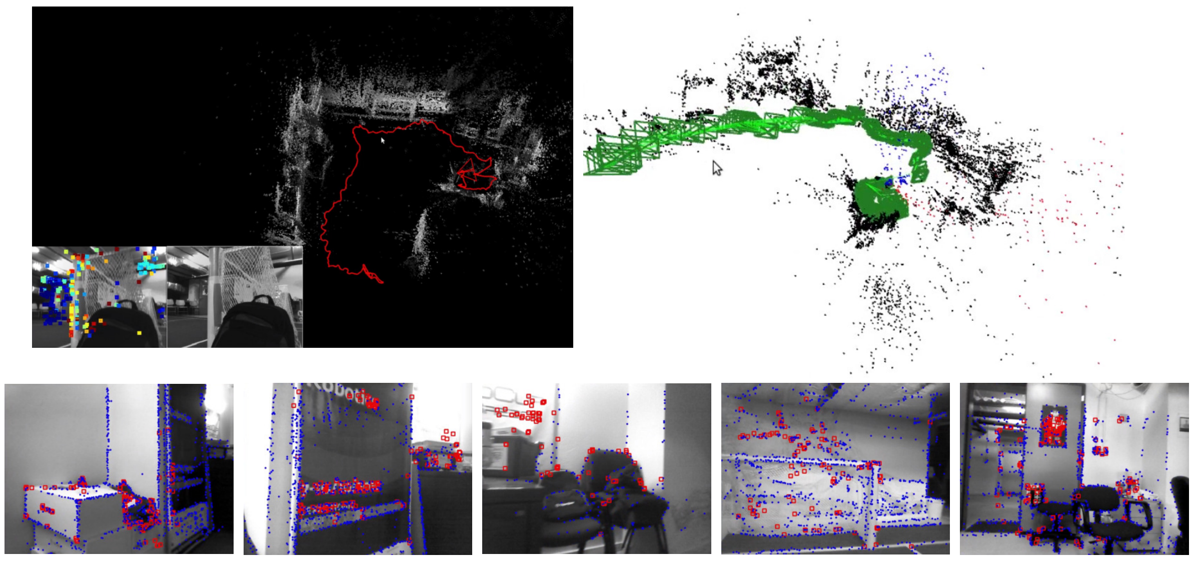


Figure 5.4: Big indoor environment reconstruction of the Kouretes Lab

5.2 Outdoor Environments

In this Section we will provide some ourdoor results on the presented in Chapter 4 monocular Visual SLAM method. In contrast with the indoor environments, the outdoor ones usually contain bigger brightness variations and more textured scenes. That means that the camera tracking can become more difficult, but a lot of points can be extracted, leading to richer scene representation with a smooth distriburion of points in the oucome point cloud map.

Scenario no1: We test the coupled system as we are heading up a stairway at the ECE department. This is a special case, as the keyframes must be quickly generated in order to correctly represent the difference in elevation we create by moving upwards. As shown in Figure 5.5, the camera trajectory can accurately describe this motion with the mapped points providing a visual understanding of the explored scenes.

Scenario no2: Nao robot tries to map an outdoor amphitheater that has a lot of texture, meaning that we can extract many points from the scenes that observe it. As shown in Figure 5.6 at the middle image of the first row, the generated point cloud map was initially corrupted as the drift during the exploration had accumulated, causing the same part of the map being rewritten above itself. However, a loop was then detected at that point as shown with the green covisibility links and a global Bundle Adjustment was

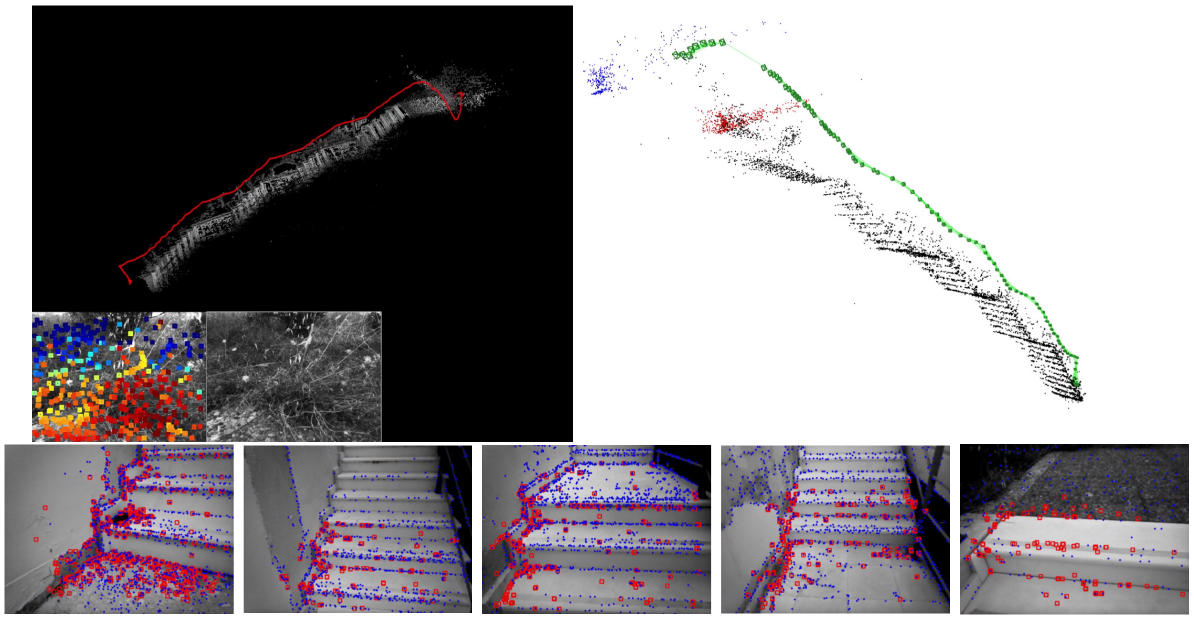


Figure 5.5: Big outdoor reconstruction of a stairway at the ECE department

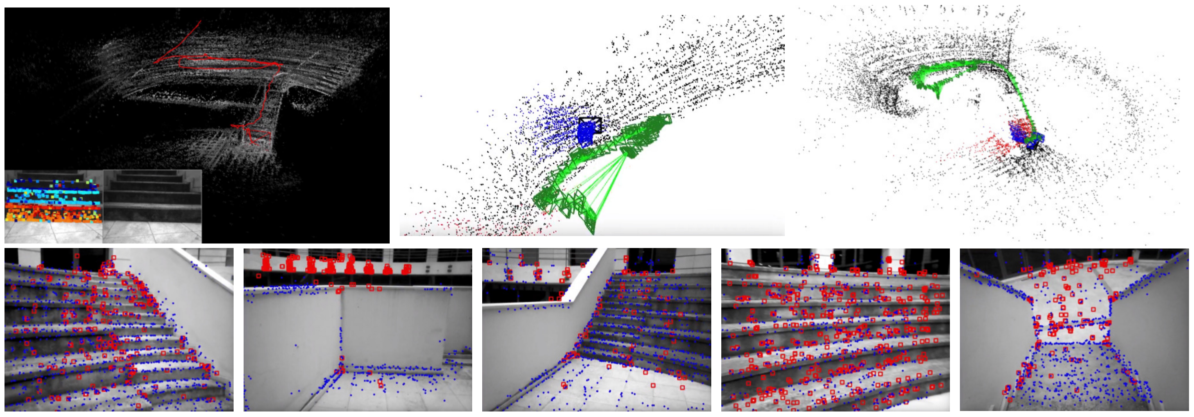


Figure 5.6: Big outdoor reconstruction of the open amphitheater at the ECE department

applied to refine the camera trajectory and the points mapped. Overall, the two edges of the covisibility graph were united to a single camera pose and the mapping continued, leading to the final map as shown at the right image of the first row.

Scenario no3: A route with many different textures was completed that also contained a stairway. As shown in Figure 5.7, DSO was able to produce a very accurate map of the environment, but the coupled system at the end of the course made some mistakes, wrongly assigning the points at the generated map. Still, the camera trajectory describes well the robot motion.

Scenario no4: The coupled method is tested at the parking of the ECE department. The brightness in this space has low values and many occlusions happen that make the Visual SLAM process difficult, as objects disappear and reappear in the scene. Figure 5.8 shows the results of this mapping that are in a high degree accurate with little drift for the camera trajectory.

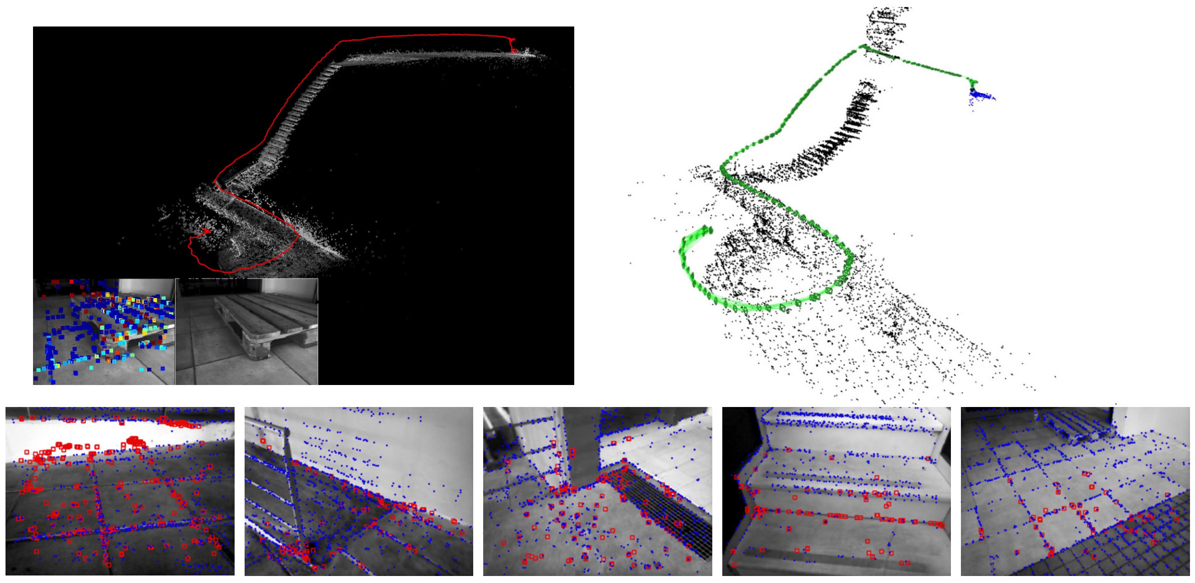


Figure 5.7: Big outdoor environment reconstruction of a complex course at the ECE department

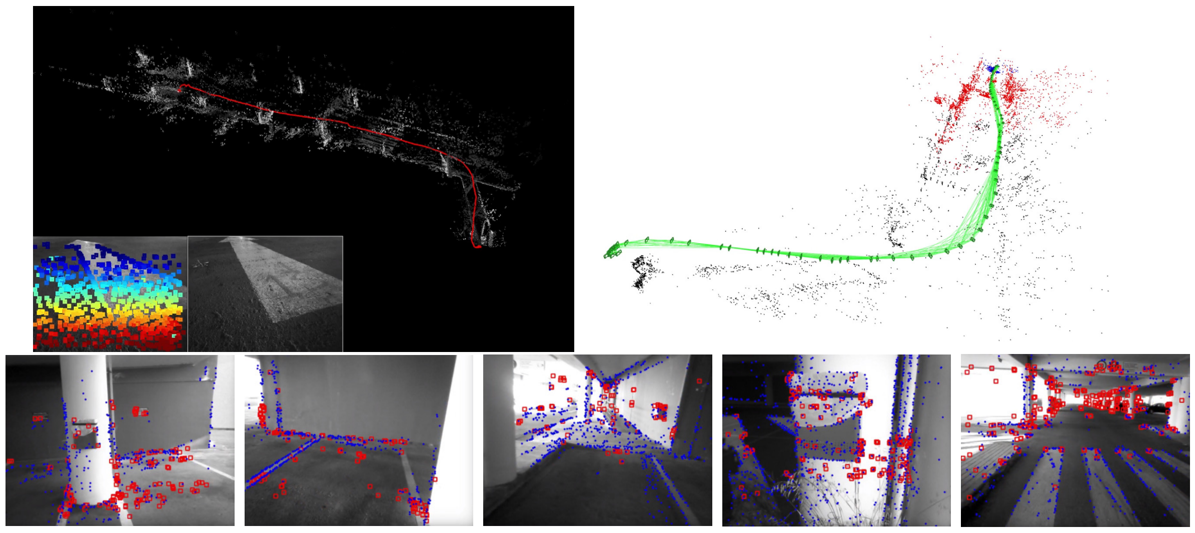


Figure 5.8: Big outdoor reconstruction of the closed parking at the ECE department

5.3 Difficulties in Monocular Visual SLAM

As we proved until now, the method presented in this thesis can provide accurate results for many environments and scenarios that can make the SLAM process of Nao difficult. However, we may still not always acquire the wanted outcomes due to the factors discussed in the beginning of this Chapter and many others, such as scene reflectance or a system module's bad estimation. In this Section we will describe some cases with corrupted maps created through exploration.

Case no1: Nao robot tries to map a classroom of the ECE department as a big indoor space. The brightness in this environment is mostly constant, but a lot of reflections exist that played a big role to the mapping failure. As shown in Figure 5.9, the DSO map has accurately mapped the classroom with very small drift for the camera trajectory. However, the coupled system has completely failed at representing the environment even though the camera trajectory seems accurate. A process that could correct this error is the loop closure that would lead to a pose graph optimization, but it was not detected in this case.

Case no2: Another case that can cause problems to the final map is the wrong scale estimation. This is usually caused because of bad system initialization and exists exclusively for monocular camera setups. In Figure 5.10 and 5.11 we can see two examples of such a mapping, where the camera poses are correctly tracked through Nao's video sequence, but the scale that describes the relation in depth between the camera pose and

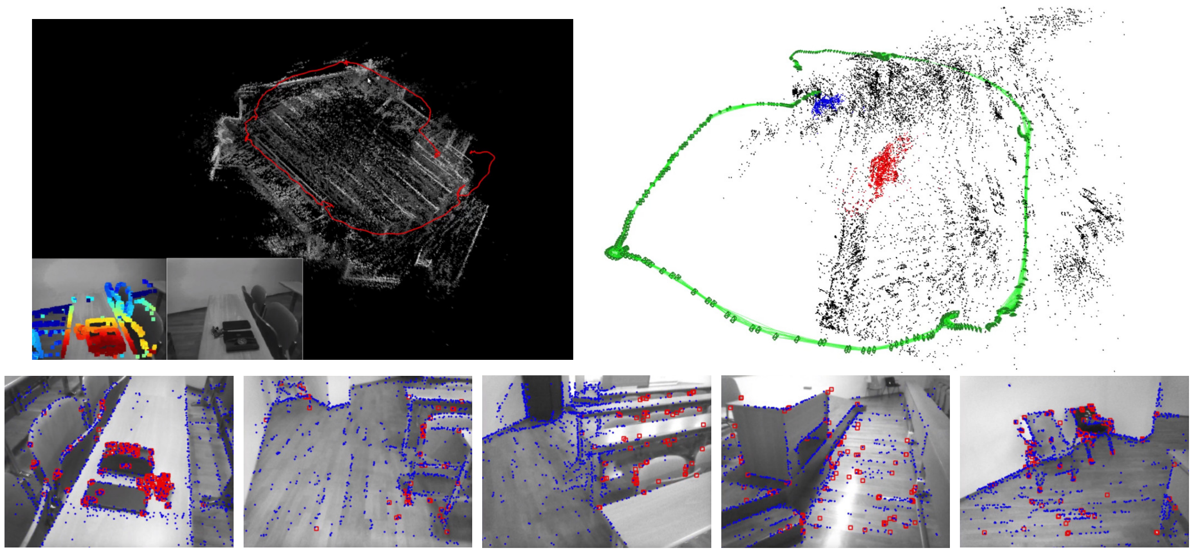


Figure 5.9: Bad map generation of a big indoor environment due to drift and light reflections

the environment is completely off.

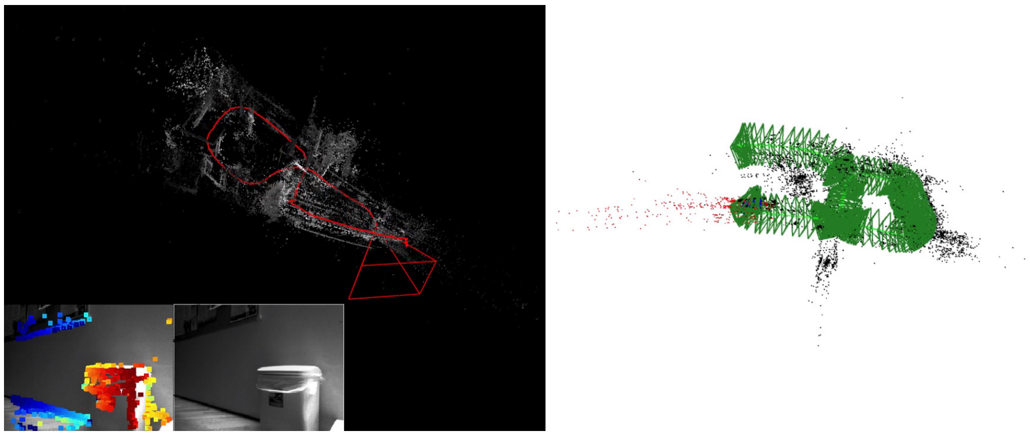


Figure 5.10: Bad map generation of a big indoor environment due to wrong scale estimation at the ECE department

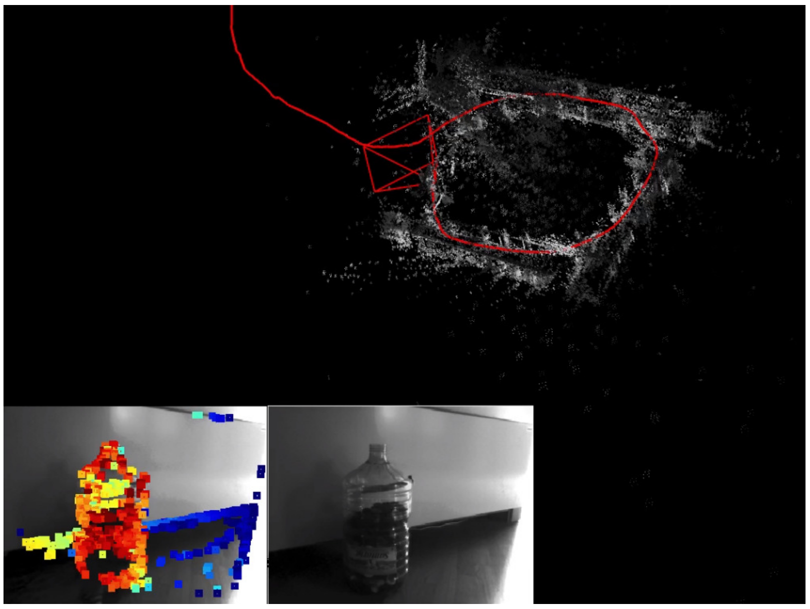


Figure 5.11: Bad map generation of the ECE building's core due to wrong scale estimation

Chapter 6

Conclusion

6.1 Summary

This thesis describes a coupled semi-direct visual Simultaneous Localization and Mapping (SLAM) approach for the Nao humanoid robot. Both subsystems chosen are the best performing choices from the respective group of methods up to this day, with the Direct module used for local camera tracking and system initialization, and the Feature-based module for local area consistency, loop closing, and global level map refinement. Furthermore, we are exploiting the capabilities of the Robot Operating System (ROS), which in the proposed approach supports not only the communication between the two subsystems, but also the robot-to-computer message transmission. In total, all the computational load is being managed by a remote computer station in a highly-parallel architecture for greater efficiency and real-time system operation. The proposed method is suitable for the monocular camera setup of Nao, where depth can only be estimated up to a scale. Last but not least, a teleoperation node that works in parallel with the overall system is built in order to simulate real-life autonomous robot exploration.

6.2 Future Work

6.2.1 Effective Semi-Direct Visual Odometry Coupling

The method used in this thesis can be described as a loose coupling of the chosen DSO and ORB-SLAM systems. That is because both maps are updated asynchronously to each other, and then the needed information is transmitted to the final system. To reduce the total unnecessary computation load, a more effective approach than the current loosely-coupled one can be introduced. The idea is that instead of maintaining each subsystem's map in real-time, we can directly use the modules of each subsystem we are interested in on the final map, with all the computations and optimizations dealing with the camera poses and extracted points, happening based on that map's information.

6.2.2 Inertial Visual Odometry Measurements

The motion of Nao robot model is generally not smooth when it comes to camera measurements and can lead to very different scene views even for close in time frames. Events like that can cause the camera pose tracking procedure to fail as there are not enough points between keyframes to be matched, meaning that the total system will then terminate, making the robot unable to continue it's SLAM process. A solution to this problem is to also include the inertial measurements from the Nao robot's IMU sensor unit, to make the final camera pose estimation more accurate and robust with no great cost in computations. The idea behind Visual Inertial Odometry is used by many published robotic systems today showing good results and is briefly discussed by us in Section 2.4.4.

A significant amount of work has been put for such a system in this thesis but was not combined with the total coupled Hybrid Visual SLAM approach we presented. Briefly, we can describe the overall rotation that concerns the top camera frame, by combining rotations between teams of coordinate frames inside the Nao robot model as shown in Figure 4.35. The first distinct rotation is created between the world coordinate frame and the Nao torso frame, the second between the Neck and the Head frame, while the third between the Head and the camera top frame. All of these rotations are translated to the top camera frame by using the tf ROS library [51]. The visualization is also possible by using the Rviz ROS tool [52]. In Figure 6.1 we show how the problem is broken down to using just the essential coordinate frames. We must also note that the information about the robot motion, described by the relation between the world coordinate frame and the torso frame, comes from the */naoqi_driver/odom* ROS topic.

This procedure will provide us in real-time with a camera pose that is mostly based on raw data and can then be further estimated by taking into account measurements from

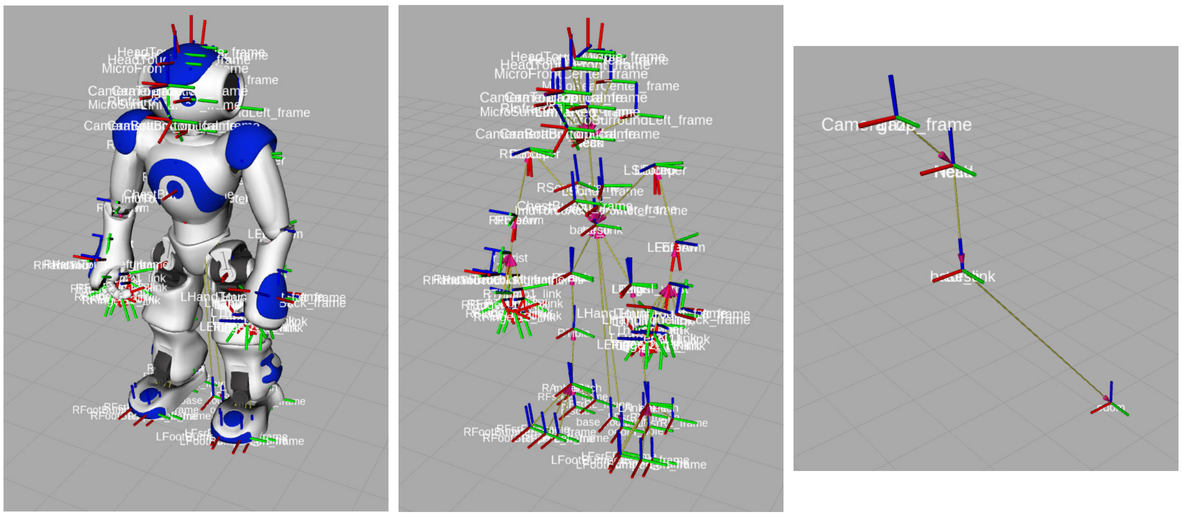


Figure 6.1: From left to right, the total coordinate frames with the Nao robot model, in the middle only the total coordinate frames and then just the essential for the top camera frames

the Nao's IMU. A package that is compatible with the ROS framework and can combine different sensor information to approximate the robot pose inside an EKF framework is [53]. At last, the final camera pose that is computed as discussed in this Section, can be given to the coupled Hybrid Visual SLAM system to refine its own estimation in order to make the camera tracking module more effective and robust.

6.2.3 Online Photometric Calibration

In our approach we include an offline photometric calibration for the monocular camera setup of Nao robot. Overall, this calibration can describe the response camera function and the vignette function of our camera. These estimations can be done as we have access to the exposure times of Nao's camera during the system's execution. In case the robot navigates between environments with very different brightness in them and an auto-exposure algorithm is applied to the camera, an online photometric calibration can be proposed to dynamically estimate the response, vignette, and exposure times. This provides greater autonomy and can further generalize our approach for application to other robot models too. A modern work that introduces such a method is [54] of

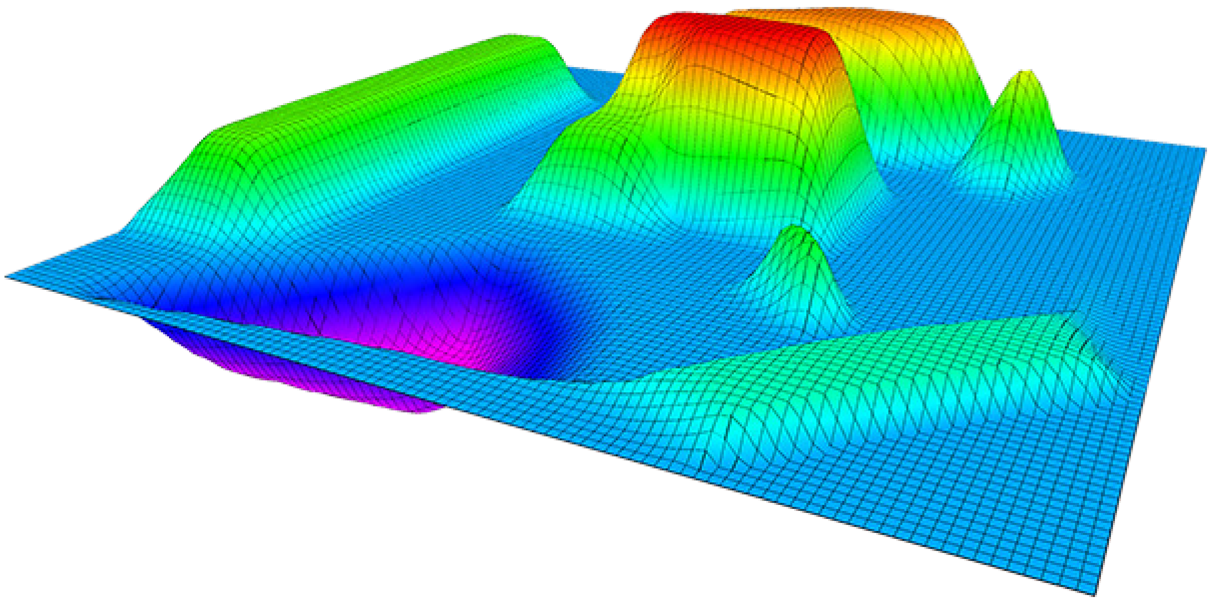


Figure 6.2: A 2.5 dimensional map generated by the *Grid Map* ROS package

Bergmann, Wang, and Cremers, which share the code as open-source[1].

6.2.4 Grid Map Generation

In order for Nao to conduct a really autonomous navigation task, it needs to be provided with an easily readable map. However, the Point Cloud map that we generate in this thesis may contain outliers or noisy gathered point locations that can not be well translated to space allocation. A way to solve this problem is to convert the Point Cloud map into an Occupancy Grid map that segmentates the 3D environment around the robot with grids assigning them with a probability to be occupied. An example work that is also compatible with the ROS message protocol is the *Grid Map*[2] package that can develop a 2.5 dimensional grid map from a 3D Point Cloud map as discussed in [55]. A converted map is provided in Figure 6.2 as shown in the package's documentation. These kind of maps are ideal for autonomous navigation, as the robot can in a simple way distinguish different locations in the map and access them by using path-planning algorithms.

[1] https://github.com/tum-vision/online_photometric_calibration
[2] https://github.com/ANYbotics/grid_map

References

[1] Lee, S.H., Civera, J.: Loosely-coupled semi-direct monocular slam. IEEE Robotics and Automation Letters **4**(2) (2018) 399–406 2, 118

[2] Engel, J., Koltun, V., Cremers, D.: Direct sparse odometry. IEEE transactions on pattern analysis and machine intelligence **40**(3) (2017) 611–625 2, 20, 94

[3] Mur-Artal, R., Montiel, J.M.M., Tardos, J.D.: Orb-slam: a versatile and accurate monocular slam system. IEEE transactions on robotics **31**(5) (2015) 1147–1163 2, 69, 104

[4] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. Volume 3., Kobe, Japan (2009) 5 3, 8, 73

[5] Thrun, S., Burgard, W., Fox, D.: Probabilistic robotics. MIT press (2005) 12

[6] Engel, J., Usenko, V., Cremers, D.: A photometrically calibrated benchmark for monocular visual odometry. arXiv preprint arXiv:1607.02555 (2016) 20, 81, 83, 104, 127

[7] Cremers, D.: Lectures on multiple view geometry (July 2014) 23

[8] Devernay, F., Faugeras, O.D.: Automatic calibration and removal of distortion from scenes of structured environments. In: Investigative and Trial Image Processing. Volume 2567., International Society for Optics and Photonics (1995) 62–73 38

[9] Stein, G.P.: Lens distortion calibration using point correspondences. In: Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, IEEE (1997) 602–608 39

[10] Fitzgibbon, A.W.: Simultaneous linear estimation of multiple view geometry and lens distortion. In: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001. Volume 1., IEEE (2001) I–I 39

[11] Zhang, Z.: A flexible new technique for camera calibration. IEEE Transactions on pattern analysis and machine intelligence **22** (2000) 41, 43, 76

[12] Hartley, R., Zisserman, A.: Multiple view geometry in computer vision. Cambridge university press (2003) 41

[13] Gavin, H.: The levenberg-marquardt method for nonlinear least squares curve-fitting problems. Department of Civil and Environmental Engineering, Duke University (2011) 1–15 43, 80, 87

[14] Heikkila, J., Silven, O., et al.: A four-step camera calibration procedure with implicit image correction. In: cvpr. Volume 97. (1997) 1106 44

[15] Grossberg, M.D., Nayar, S.K.: What is the space of camera response functions? In: 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings. Volume 2., Citeseer (2003) II–602 45

[16] Grossberg, M.D., Nayar, S.K.: Determining the camera response from images: What is knowable? IEEE Transactions on pattern analysis and machine intelligence **25**(11) (2003) 1455–1467 47

[17] Lucas, B.D., Kanade, T., et al.: An iterative image registration technique with an application to stereo vision. (1981) 51

[18] Rosten, E., Drummond, T.: Machine learning for high-speed corner detection. In: European conference on computer vision, Springer (2006) 430–443 54, 70

[19] Harris, C.G., Stephens, M., et al.: A combined corner and edge detector. In: Alvey vision conference. Volume 15., Citeseer (1988) 10–5244 55

[20] Shi, J., Tomasi, C.: Good features to track. Technical report, Cornell University (1993) 55

[21] Klein, G., Murray, D.: Parallel tracking and mapping for small ar workspaces. In: Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality, IEEE Computer Society (2007) 1–10 69

[22] Pirker, K., Rüther, M., Bischof, H.: Cd slam-continuous localization and mapping in a dynamic world. In: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE (2011) 3990–3997 69

[23] Davison, A.J., Reid, I.D., Molton, N.D., Stasse, O.: Monoslam: Real-time single camera slam. IEEE Transactions on Pattern Analysis & Machine Intelligence (6) (2007) 1052–1067 69

[24] Ranftl, R., Vineet, V., Chen, Q., Koltun, V.: Dense monocular depth estimation in complex dynamic scenes. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2016) 4058–4066 69

[25] Rublee, E., Rabaud, V., Konolige, K., Bradski, G.R.: Orb: An efficient alternative to sift or surf. In: ICCV. Volume 11., Citeseer (2011) 2 70, 105

[26] Bay, H., Tuytelaars, T., Van Gool, L.: Surf: Speeded up robust features. In: European conference on computer vision, Springer (2006) 404–417 70

[27] Newcombe, R.A., Lovegrove, S.J., Davison, A.J.: Dtam: Dense tracking and mapping in real-time. In: 2011 international conference on computer vision, IEEE (2011) 2320–2327 70

[28] Engel, J., Schöps, T., Cremers, D.: Lsd-slam: Large-scale direct monocular slam. In: European conference on computer vision, Springer (2014) 834–849 70, 72

[29] Forster, C., Pizzoli, M., Scaramuzza, D.: Svo: Fast semi-direct monocular visual odometry. In: 2014 IEEE international conference on robotics and automation (ICRA), IEEE (2014) 15–22 71

[30] Li, S.p., Zhang, T., Gao, X., Wang, D., Xian, Y.: Semi-direct monocular visual and visual-inertial slam with loop closure detection. Robotics and Autonomous Systems **112** (2019) 201–210 71

[31] Mei, C., Sibley, G., Cummins, M., Newman, P., Reid, I.: Rslam: A system for large-scale mapping in constant-time using stereo. International journal of computer vision **94**(2) (2011) 198–214 71

[32] Pire, T., Fischer, T., Civera, J., De Cristóforis, P., Berlles, J.J.: Stereo parallel tracking and mapping for robot localization. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE (2015) 1373–1378 71, 72

[33] Engel, J., Stückler, J., Cremers, D.: Large-scale direct slam with stereo cameras. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE (2015) 1935–1942 71, 72

[34] Newcombe, R.A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A.J., Kohli, P., Shotton, J., Hodges, S., Fitzgibbon, A.W.: Kinectfusion: Real-time dense surface mapping and tracking. In: ISMAR. Volume 11. (2011) 127–136 72

[35] Whelan, T., Salas-Moreno, R.F., Glocker, B., Davison, A.J., Leutenegger, S.: Elasticfusion: Real-time dense slam and light source estimation. The International Journal of Robotics Research **35**(14) (2016) 1697–1716 72

[36] Knese, K.: Driver module between aldebaran's naoqios and ros. (2010) 74, 120

[37] James Bowman, P.M.: A ros package for camera calibration using the opencv library. (2014) 76

[38] Garrido-Jurado, S., Muñoz-Salinas, R., Madrid-Cuevas, F.J., Marín-Jiménez, M.J.: Automatic generation and detection of highly reliable fiducial markers under occlusion. Pattern Recognition **47**(6) (2014) 2280–2292 86

[39] Munoz-Salinas, R., Garrido-Jurado, S.: Aruco library. URL: http://sourceforge.net/projects/aruco (2013) 87

[40] Civera, J., Davison, A.J., Montiel, J.M.: Inverse depth parametrization for monocular slam. IEEE transactions on robotics **24**(5) (2008) 932–945 90

[41] Civera, J., Davison, A.J., Montiel, J.M.M.: Inverse depth to depth conversion for monocular slam. In: Proceedings 2007 IEEE International Conference on Robotics and Automation, IEEE (2007) 2778–2783 90

[42] Montiel, J.M., Civera, J., Davison, A.J.: Unified inverse depth parametrization for monocular slam, Robotics: Science and Systems (2006) 90

[43] Mur-Artal, R., Tardós, J.D.: Orb-slam: Tracking and mapping recognizable. In: Workshop on Multi View Geometry in Robotics (MVIGRO)-RSS. (2014) 104

[44] Mur-Artal, R., Tardós, J.D.: Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. IEEE Transactions on Robotics **33**(5) (2017) 1255–1262 104

[45] Viswanathan, D.G.: Features from accelerated segment test (fast) (2009) 105

[46] Calonder, M., Lepetit, V., Strecha, C., Fua, P.: Brief: Binary robust independent elementary features. In: European conference on computer vision, Springer (2010) 778–792 105

[47] Strasdat, H., Davison, A.J., Montiel, J.M., Konolige, K.: Double window optimisation for constant time visual slam. In: 2011 International Conference on Computer Vision, IEEE (2011) 2352–2359 109

[48] Gálvez-López, D., Tardos, J.D.: Bags of binary words for fast place recognition in image sequences. IEEE Transactions on Robotics **28**(5) (2012) 1188–1197 109

[49] Mur-Artal, R., Tardós, J.D.: Fast relocalisation and loop closing in keyframe-based slam. In: 2014 IEEE International Conference on Robotics and Automation (ICRA), IEEE (2014) 846–853 115

[50] Yang, N., Wang, R., Gao, X., Cremers, D.: Challenges in monocular visual odometry: Photometric calibration, motion bias, and rolling shutter effect. IEEE Robotics and Automation Letters **3**(4) (2018) 2878–2885 118

[51] Foote, T.: tf: The transform library. In: 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), IEEE (2013) 1–6 144

[52] Kam, H.R., Lee, S.H., Park, T., Kim, C.H.: Rviz: a toolkit for real domain data visualization. Telecommunication Systems **60**(2) (2015) 337–345 144

[53] Meeussen, W.: A ros package for robot pose estimation by combining many sensor measurements inside an ekf framework 145

[54] Bergmann, P., Wang, R., Cremers, D.: Online photometric calibration of auto exposure video for realtime visual odometry and slam. IEEE Robotics and Automation Letters **3**(2) (2017) 627–634 145

[55] Fankhauser, P., Hutter, M.: A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation. In Koubaa, A., ed.: Robot Operating System (ROS) The Complete Reference (Volume 1). Springer (2016) 146