Ferry-SLAM: A System for Environment Perception and Motion Estimation for the Autonomous Ferry

Vipul Garg

Master Thesis

Erasmus Mundus Master in
Marine and Maritime Intelligent Robotics

Universitat Jaume I

September 18, 2023

Supervised by: Prof. Enric Cervera Mateu
Co-supervised by: Prof. Rudolf Mester

Co-funded by the
Erasmus+ Programme
of the European Union

Acknowledgments

I am deeply grateful to several individuals and organizations whose invaluable contributions and unwavering support have made the completion of this thesis possible.

First and foremost, I extend my sincere appreciation to Prof. Enric Cervera Mateu for his guidance, mentorship, and willingness to supervise my research. His acceptance of my chosen topic provided me with the opportunity to embark on this academic journey.

I am indebted to Prof. Rudolf Mester for his role as a co-supervisor. His steadfast support and guidance throughout the thesis period were instrumental in shaping the direction of my research. His insightful advice and encouragement propelled me forward at every stage.

I extend my gratitude to the members of the Stereo-Vision Group: Trym Anthonsen Nygård, Nicholas Dalhaug, and Prof. Annette Stahl. Our regular meetings and knowledge-sharing sessions were invaluable in refining my ideas and enhancing my understanding of the subject matter.

My heartfelt thanks go to Zeabuz for hosting me during the course of this thesis and for providing invaluable industrial insights. The support and resources provided by the company were essential in bridging the gap between academic theory and practical application. I also wish to acknowledge Dr. Øystein K. Helgesen and the SitAW team at Zeabuz for their valuable insights and contributions, which enriched the depth of my research.

Finally, I express my deep appreciation to my brother for his unwavering motivation and encouragement throughout the entirety of this journey. His belief in my abilities was a constant source of inspiration.

To all those mentioned above and to countless others who have supported me in various ways, I offer my heartfelt thanks. Your contributions have shaped not only this thesis but also my growth as a researcher.

Abstract

This thesis addresses the application of stereo cameras in the context of maritime vessel localization and environmental perception. The focus is on developing an independent vision-based system for localization and perception, applicable to various maritime and ground vehicles. The research is conducted in collaboration with Universitat Jaume I (UJI), Norwegian University of Science and Technology (NTNU), and Zeabuz, a startup specializing in autonomous ferries.

The project centers around the MilliAmpere-2, an electric research vessel equipped with LiDARs, cameras, radar, and a dynamic positioning system. Despite its specific application, this thesis aims to create a vision-based solution adaptable to diverse maritime and ground vehicles. Challenges such as the dynamic sea surface and limited visibility of distant structures motivate the need for a specialized vision-based system.

The proposed perception system, named Ferry-SLAM, integrates visual odometry and environmental representation. Key contributions include masking strategies for keypoint generation, predictive keypoint matching, novel techniques for rotational estimation from distant regions, the fusion of semantic segmentation and 3D plane fitting, and a unique approach to distinguishing upright and horizontal structures. The system employs bird's-eye views for navigation, enhancing efficiency without sacrificing information.

Ferry-SLAM is designed to address the complex maritime environment's challenges, including the dynamic sea surface and limited visibility. Semantic segmentation and 3D plane fitting provide a robust understanding of the environment, aiding in navigation and obstacle avoidance. The innovative rotational estimation methods enhance accuracy, and predictive keypoint matching ensures reliable correspondence estimation.

The developed perception system is evaluated using datasets, including the KITTI dataset and locally recorded sequences. Results indicate promising accuracy in trajectory estimation, even when facing maritime-specific challenges. This work contributes to the field of maritime and ground vehicle autonomy by offering a vision-based solution tailored to the unique demands of open-water navigation.

Keywords— ego-motion estimation, scene analysis, object detection, keypoints, correspondence matching, semantic segmentation, bird's eye view.

Contents

Abbreviations

AI Artificial Intelligence

APSR Adaptive Plane Segmentation using RANSAC

BEV Bird's Eye View

BM Block Matching

CCF Camera Coordinate Frame

CorrEst Correspondence Estimator

CV Computer Vision

FOV Field of View

GFTT Good Features to Track

GUI Graphical User Interface

IMOs Independent Moving Objects

KptDet Keypoints Detector

LiDAR Light Detection and Range

LK Lukas-Kanade

NN Neural Network

PhC phase correlation

PnP Perspective-n-Point

RANSAC Random sample consensus

ROI Region of Interest

SGBM Semi Global Block Matching

SLAM Simultaneous Localization and Mapping

SSD Sum of Squared Differences

SVD Singular Value Decomposition

VO Visual Odometry

V-SLAM Visual Simultaneous Localization and Mapping (SLAM)

WCF World Coordinate Frame

w.r.t. with respect to

Part I

Introduction and Overview

CHAPTER 1

INTRODUCTION

Contents

This project is about the usage of stereo cameras in the context of the localization of the maritime vessel and the perception of the surrounding environment. This work applies the principles of the Visual Odometry (VO) to estimate the ego-motion of the maritime vessel called MilliAmpere-2 ([8]). The MilliAmpere-2 (see Fig. 1.1) is a research vessel developed by the Norwegian University of Science and Technology (NTNU). It is an electric boat in general but equipped with Light Detection and Range (LiDAR)s, cameras, one RADAR, etc. for the perception of the environment and the dynamic positioning system for the control of the boat. The working principles of the MilliAmpere-2 are outside the scope of this work because this project is intended to develop an independent vision-based system that can be integrated into any maritime vessel irrespective of the other sensors. The focus of this project is not limited to only the maritime vessels but also extended to the ground vehicles as the problem statement remains the same.

The work was done between $Feb.$, 2023 and $July$, 2023 in collaboration with the Universitat Jaume I (UJI), NTNU, and Zeabuz. Zeabuz is a startup based on autonomous ferries and it is the industrial partner of this project. As two of the three partners are based in Trondheim, Norway, the work has also been carried out in Trondheim. In the following sections, I will talk about the context and my contribution to this project and finally give an outline of the thesis.

1.1 Context of this work and Motivation

Localization and navigation have been one of the biggest challenges in the development of autonomous cars by the automotive industries. On top of that, the perception of the environment

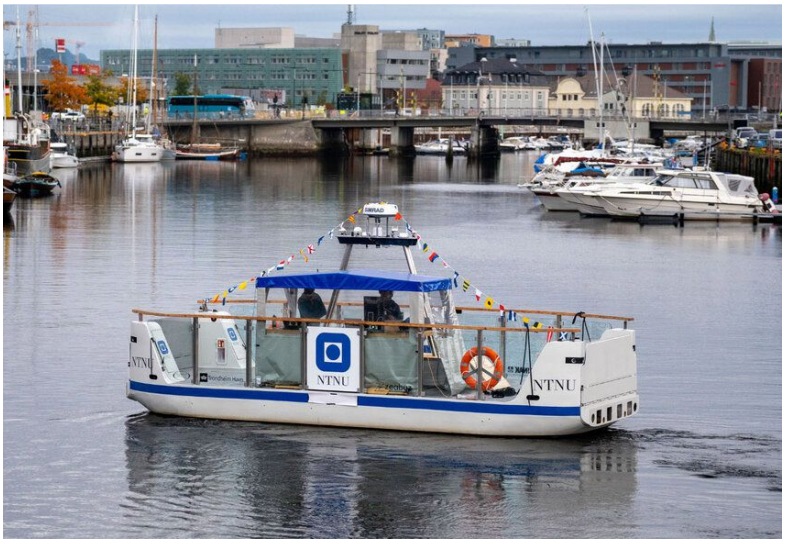


Figure 1.1: MilliAmpere 2. Taken from [23].

makes this problem even more complicated. Over the decade, multiple attempts have been made in this domain by different research and industrial institutions. When we talk about localization in autonomous systems, one word usually comes in the context and that is called Simultaneous Localization and Mapping (SLAM). The SLAM is a framework that allows the user to build the map of the environment and localize itself in it simultaneously. The SLAM is a concept that can be implemented in different ways. The initial proposal of the SLAM was based on the use of the LiDARs to scan the environment with precise range measurements and then identify the landmarks in the environment such that they can be tracked over time and can be used to build the map. With the advancements in the Computer Vision (CV), the use of the cameras in the SLAM became an active research topic as the images carry more information than the LiDARs and they are cheap to equip the robot with. Also, they can be very helpful in the perception of the environment which was not completely possible by the LiDARs only.

The Visual SLAM (V-SLAM) has matured over the years, particularly in the automotive industry. The V-SLAM system has been favored in the automotive sector because of the need for high precision in the localization in the narrow streets but the question *"Do we need SLAM for the maritime vessels?"* needs to be answered as the vessels operate mostly in the open water and the approximate position from the GPS is enough to localize and navigate. The answer to this question is *Yes* as we need such a system because when the vessel arrives at the harbor or passes under the bridge, the need for the exact increases because the position from the GPS is not reliable in such scenarios. Now, the second question is *"Can we use the already developed SLAM systems on the vessels?"*. The most common SLAM approaches can not be applied to maritime vessels directly because of the following challenges.

- The dynamic sea surface is one of the biggest challenges. The ego-vessel always stays in motion because of this non-rigid water surface and makes the problem from the constrained planar surface problem to the open 3D space problem. Also, in the keypoint based V-SLAM system, the keypoints identified on the sea can't be considered landmarks or unique keypoints because of the water dynamics.
- The limited visibility of the far-located structures presents the next challenge. When

the ego-vessel is far from the harbor, the sky, and the sea dominate the image, and the buildings, harbor, etc. have a very small coverage in the image view. The identification and tracking of the keypoints become more complicated because of the limited resolution of the camera.

The above challenges are the tip of the iceberg that the maritime vessel has to overcome. Apart from the mentioned challenges, the identification and the tracking of other boats are also equally important because they are not only potential obstacles that the ego-vehicle can collide with but they can also bias the ego-motion estimation.

1.2 Contributions

I developed a perception system for maritime vehicles, called Ferry-SLAM that does not only include a VO subsystem, but also a subsystem for creating representations of the environment from single stereo image pairs. In this system, I proposed and developed multiple innovative methods and techniques related to scene representation, object detection, and motion estimation, and a few of them are listed below.

- **Motion estimation:**
  - **Masking of the image:** I introduced a masking strategy that can be used to highlight the areas of the images that should be avoided for the generation of the keypoints.
  - **Predictive Keypoint Matching:** Instead of using the conventional way of finding the correspondences of the keypoints in one step, I used predictive keypoint matching to reduce the chances of the occurrences of the outliers.
  - **Rotation from far areas:** I proved that the 2D motion of the distant far regions in the image happens only due to the rotation and then I derived the expressions to compute the rotation from the distant far regions and proposed three different techniques to benefit from this principle.
  - **Profile-based rotation estimation:** It is one of the novel techniques I developed to estimate the rotation from far areas. It uses depth information to prepare the image signal and then estimates the change of the yaw angle between two camera frames.
- **Scene representation and object detection:**
  - **Fusion of semantic segmentation and 3D plane fitting:** I developed a method to extract the ground plane using an adaptive 3D plane fitting algorithm and fused the 3D plane information with the semantic labels generated from two different semantic segmentation networks to perceive the maritime environment.
  - **Detection of upright and horizontal structures:** I also proposed an algorithm that distinguishes the upright and horizontal structures using the disparity data only. It is another method to perceive the environment and to find the corridor in which the maritime vessel can freely operate.
  - **Bird's Eye View (BEV)s for the navigation:** Instead of using the 3D point clouds, I used the BEVs to represent the 3D scene in 2D without losing the most important information that is required for the navigation.

- **Interactive sequencer:** I built an interactive sequencer (dataset handler) that is very useful for the debugging of any CV based application. It handles multiple types of datasets in the background and provides a Graphical User Interface (GUI) for the interaction with the sequencer and the application.

1.3 Outline of the thesis

The work done in this project is split into 6 parts of the thesis. Part I introduces the objective and motivation of this project as well as gives an overview of my contributions to it.

Part II deals with the revision of the fundamentals required for the project and discusses the requirements to develop the Ferry-SLAM system such as the dataset to test the system. In chapter 2, I have explained the geometrical concepts for the Computer Vision (CV). It includes the pinhole camera model, an introduction to the camera calibration matrices, the stereo camera, etc. In chapter 3, the fundamental concepts required for motion estimation such as scene flow, Visual Odometry (VO), keypoint matching algorithms, etc. are presented. Finally, in chapter 4, I am talking about the KITTI dataset and my own dataset that I recorded for the development of the Ferry-SLAM system.

In Part III, I gave an overview of the Ferry-SLAM followed by its modules required for maritime scene perception. In chapter 5, I have presented the whole architecture of the Ferry-SLAM system. It includes the modules responsible for scene perception, scene representation, and ego-motion estimation. In chapter 6, I presented the features of the tool that I developed for the smooth development of the system. It includes a sequencer that handles the dataset and a GUI. In chapter 7, two different semantic segmentation-based networks have been discussed along with the geometry-based ground plane identifier such that their information can be fused together to perceive the scene. In chapter 8, another method to perceive the scene has been presented that distinguishes different structures based on their disparity profiles. In chapter 9, I have explained the BEV and its importance in representing the scene and highlighted its suitability for navigation.

Part IV explains the ego-motion estimation sub-system and its components in detail. In chapter 10, I provided an overview of its modules. In chapter 11, an overview of different methods to predict the pose is given and its importance and implementation in my project have been highlighted. In chapter 12, I explained the strategy to generate the keypoints in the image followed by the algorithm to find the correspondences of the keypoints in chapter 13. In chapter 14, I explained the principle behind the estimation of the rotation from far-away regions and the derivations involved. I also presented three different techniques to estimate the 2D motion of distant far regions followed by the estimation of the rotation. In chapter 15, I talked about the strategy to initialize the system and the optimizer used to estimate the motion from the correspondences. In chapter 16, I proposed an idea for future work that can make the ego-motion estimation sub-system more robust and stable with respect to the outliers in the correspondences.

Part V deals with the experiments and the results. In chapter 17, I analyzed the ego-motion estimator and checked if some reconfigurations are required in the system or not, and in chapter 18, I plotted the trajectory generated using the ego-motion estimator and commented on the results.

In part VI, I will conclude the thesis with the conclusion (see chapter 19, p.163) and present the work that needs be to done to continue the project in the future (see chapter 20, p.165).

Part II

Background of the **Ferry-SLAM** Project

CHAPTER 2

Theoretical Background on Geometric Computer Vision

Contents

The Geometric Computer Vision (CV) is a sub-domain of CV that deals with the intricate spatial characteristics embedded within the visual data. This chapter revises the fundamental concepts required to understand the core modules of the Ferry-SLAM system. It includes the definition and notation of the *pose*, the *camera model*, the calibration parameters of a monocular camera followed by the introduction to the stereo camera, and the *stereo-matching* algorithm.

2.1 Pose definition and notation

The main objective of any Visual Odometry (VO) is to recover the path incrementally, pose after pose, therefore, it is imperative to talk about the definition of the pose, and the notations used for the pose in this project before going any further. The pose represents the *position* and *orientation* of any object or any body (usually in 3 dimensions). The pose is always measured with respect to (w.r.t.) some *reference frame* and therefore, the pose of any object can also be referred to as the relative pose (pose relative to some reference frame). In CV and robotics, the *absolute pose* term is also used to see the pose of the camera attached to a body w.r.t some static or *World Coordinate Frame (WCF)*. The WCF is static and doesn't change over time. In most of the applications, the *initial pose* of the camera (the pose of the camera before the algorithm runs) is treated as the static/WCF and the objective of the algorithm becomes to

estimate the absolute pose of the camera at any instant w.r.t the initial/WCF. I followed the same terminology and assumed that the initial reference frame of the camera is the WCF.

As I stated before, the pose is composed of the position and the orientation. I used the Cartesian coordinates to represent the position and the *Euler angles* to represent the orientation. Euler angles represent the orientation of a body using three angles only. The use of the *rotation matrix* during the computations is more convenient than the Euler angles because of their unique properties such as rotation matrices can be used as operators to perform the transformations. In the project, I defined the pose as a *homogenous transformation matrix* ($\mathbf{T}$) of size 4×4. It is composed of the rotation matrix $\mathbf{R}$ of size 3×3 and the *translation vector* $\vec{t}$ of size 3×1.

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \vec{t} \\ 0_{3\times1} & 1_{1\times1} \end{bmatrix} \tag{2.1}$$

Let $\vec{p}_k = [x_k, y_k, z_k]^T$ be a 3D point w.r.t some frame reference frame k, $\mathbf{T}_k^i$ be the transformation matrix that transforms from reference frame k to another reference frame i then the same point $\vec{p}_k$ w.r.t the reference frame i ($\vec{p}_i$) can be computed using the transformation matrix $\mathbf{T}_k^i$.

$$\begin{aligned} \begin{bmatrix} \vec{p}_i \\ 1 \end{bmatrix} &= \mathbf{T}_k^i \begin{bmatrix} \vec{p}_k \\ 1 \end{bmatrix} \\ \vec{p}_i &= \mathbf{R}_k^i \vec{p}_k + \vec{t}_k^i \end{aligned} \tag{2.2}$$

2.1.1 Conversion of the relative pose to the absolute pose

The relative poses are very helpful but we are mostly interested in the absolute pose of the ego-vehicle with respect to the initial reference frame to generate the trajectory. Therefore, these relative poses may need to be mapped into the absolute poses. As I mentioned in the previous section, I regarded the initial coordinate frame as the WCF which implies that the absolute pose of the initial coordinate frame w.r.t. the WCF is *Identity*.

$$\begin{aligned} \mathbf{T}_0^w &= identity(4) \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{2.3}$$

The $\mathbf{T}_0^w$ also represents the absolute pose of the Camera Coordinate Frame (CCF) when the first frame ($t = 0$) arrives. If the relative pose $\mathbf{T}_{t-1}^t$ is transforming the CCF from timestep $t-1$ to timestep t, then the inverse of the homogenous transformation matrix need be computed to compute the absolute pose.

$$\begin{aligned} \mathbf{T}_t^{t-1} &= inv(\mathbf{T}_{t-1}^t) \\ &= \begin{bmatrix} \mathbf{R}_{t-1}^t & \vec{t}_{t-1}^{\,t} \\ 0_{1\times3} & 1_{1\times1} \end{bmatrix}^{-1} \\ &= \begin{bmatrix} {\mathbf{R}_{t-1}^t}^T & -{\mathbf{R}_{t-1}^t}^T \times \vec{t}_{t-1}^{\,t} \\ 0_{1\times3} & 1_{1\times1} \end{bmatrix} \end{aligned} \tag{2.4}$$

Finally, the relative pose $\mathbf{T}_t^{t-1}$ is multiplied with the previous absolute pose $\mathbf{T}_{t-1}^w$ to get the absolute pose $\mathbf{T}_t^w$ at timestep t. This operation can be done recursively.

$$\mathbf{T}_t^w = \mathbf{T}_0^w \mathbf{T}_1^0 \mathbf{T}_2^1 ... \mathbf{T}_{t-1}^{t-2} \mathbf{T}_t^{t-1} \tag{2.5}$$

2.2 Perspective camera model

The most common generic model of an intensity camera is the *perspective* or *pinhole model* [44]. From Fig. 2.1, let f be the focal length of the camera, $\vec{P} \overset{def}{=} [X_c, Y_c, Z_c]^T$ be the coordinates of the 3D point w.r.t the CCF, and $\vec{p} \overset{def}{=} [x, y, z]$ be the image coordinates of the 3D point $\vec{P}$ on the *image plane* then according to the pinhole model then the 3D point $\vec{P}$ can be mapped to the image coordinate $\vec{p}$ using the focal length f only.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f\frac{X_c}{Z_c} \\ f\frac{Y_c}{Z_c} \\ f \end{bmatrix} \tag{2.6}$$

The above equation is non-linear because of the $1/Z_c$ and does not preserve distances between points (not even up to a common scaling factor) or angles between lines. However, it maps lines into lines (see section 2.2.4 of [46]). Oftenly, the z coordinate of the image point $\vec{p}$ is equal to the focal length f, therefore, the image point $\vec{p}$ can be written as

$$\vec{p} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f\frac{X_c}{Z_c} \\ f\frac{Y_c}{Z_c} \end{bmatrix} \tag{2.7}$$

Projection Using Homogenous Coordinates: If the world point $\vec{P}$ and the image point $\vec{p}$ are represented by homogenous vectors then eq. 2.6 can be written in matrix form (see section 6.1 of [22]).

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} fX_c \\ fY_c \\ Z_c \end{bmatrix} = \begin{bmatrix} f & & & \\ & f & & \\ & & 1 & \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{2.8}$$

Let $\vec{X}_c \overset{def}{=} [X_c, Y_c, Z_c, 1]^T$ and $\vec{x} \overset{def}{=} [x, y, 1]^T$ be the homogeneous representation of the 3D point $\vec{P}$ and image point $\vec{p}$ w.r.t the CCF respectively, and $\mathbf{P}$ be the *projection matrix* then eq. 2.8 can be re-written as

$$\vec{x} = \frac{1}{Z_c}\mathbf{P}\vec{X}_c \tag{2.9}$$

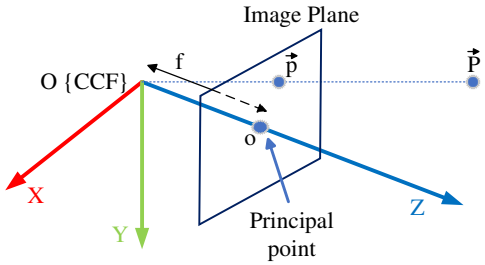


Figure 2.1: The perspective camera model.

where, $\mathbf{P} \overset{def}{=} diag(f, f, 1)[\mathbf{I}|0]$.

2.3 Camera Parameters

In *Structure From Motion* or computing the position of objects in space, we need equations that can link the 3D points from the point to the corresponding pixel coordinates of the object in the image. If the 3D point is not in the CCF, then it has to be first mapped to the CCF using the *extrinsic parameters* and then the 3D point in the CCF can be mapped to the pixel coordinates using the *intrinsic parameters*. In the following section, I will discuss these two types of camera calibration parameters in detail followed by the conversion of the pixel coordinates to the *normalized coordinates*.

2.3.1 Intrinsic Parameters

These parameters link the pixel coordinates of an image point with the corresponding 3D coordinates in the CCF or vice-versa. There are three sets of parameters needed to characterize the optics of the camera (see section 2.4.3 of [46]).

- For perspective projection, it is focal length f,
- The transformation between camera frame coordinates and pixel coordinates.
- Geometric distortion parameters such as *radial distortion*.

Principal Point Effect:
Let (p_x, p_y) be the coordinates of the principle point o w.r.t the image coordinate system (see Fig. 2.2), then the point $\vec{P}$ can be mapped to the image coordinates.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \rightarrow \begin{bmatrix} f\frac{X_c}{Z_c} + p_x \\ f\frac{Y_c}{Z_c} + p_y \end{bmatrix} \tag{2.10}$$

In homogenous coordinates, the above equation is written as

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} fX_c + Z_c p_x \\ fY_c + Z_c p_y \\ Z_c \end{bmatrix} = \begin{bmatrix} f & & p_x & 0 \\ & f & p_y & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{2.11}$$

If I assume, there is no geometric distortion and *skewness* in the image then the intrinsic parameters can be expressed in the form of a *camera calibration matrix* called $\mathbf{K}$ (see section 6.1 of [22]).

$$\mathbf{K} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.12}$$

$$\vec{x} = \frac{1}{Z_c}\mathbf{K}\begin{bmatrix} \mathbf{I} & | & 0 \end{bmatrix}\vec{X}_c \tag{2.13}$$

Further, there is a possibility of having non-square pixels in most *Charged-Couple Device (CCD)* cameras. If image coordinates are measured in pixels, then this has the extra effect of introducing

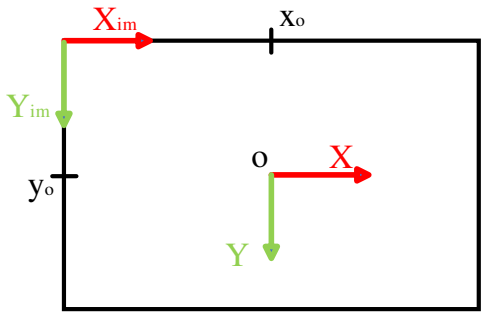


Figure 2.2: Image (x_{im}, y_{im}) and camera (x, y) coordinate system.

unequal scale factors in each direction. Let m_x and m_y be the number of pixels per unit distance in the $x-axis$ and $y-axis$ of the image coordinate system, and (x_o, y_o) be the pixel coordinates of the principle point o then the camera calibration matrix $\mathbf{K}$ also depends on the scaling factors m_x and m_y.

$$\begin{aligned}\mathbf{K} &= \begin{bmatrix} m_x & 0 & 0 \\ 0 & m_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} fm_x & 0 & m_x p_x \\ 0 & fm_y & m_y p_y \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} f_x & 0 & x_o \\ 0 & f_y & y_o \\ 0 & 0 & 1 \end{bmatrix}\end{aligned} \tag{2.14}$$

2.3.2 Extrinsic Parameters

The extrinsic parameters define the pose (position and orientation) of the camera w.r.t some know frame such as world frame (see section 2.4.2 of [46]).

Let $\vec{X}_w \overset{def}{=} [X_w, Y_w, Z_w, 1]^T$ be the homogenous coordinates of the 3D point $\vec{P}$ w.r.t the WCF, $\mathbf{R}$ be the rotation matrix representing the orientation of the CCF, and $\vec{c}$ be the translation vector that represents the coordinates of the camera center in the WCF, then the point $\vec{X}_w$ w.r.t WCF can be mapped to the point $\vec{X}_c$ w.r.t the CCF using the rotation matrix $\mathbf{R}$ and the translation vector $\vec{c}$ (ref section 6.1 of [22]),

$$\vec{X}_c = \begin{bmatrix} \mathbf{R} & -\mathbf{R}\vec{c} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.15}$$

The above equation can be reduced to

$$\vec{X}_c = \mathbf{R} \begin{bmatrix} \mathbf{I} & -\vec{c} \\ 0 & 1 \end{bmatrix} \vec{X}_w \tag{2.16}$$

It is often convenient not to make the camera center explicit and instead to represent the world to image transformation as $\vec{t}$.

$$\vec{X}_c = \begin{bmatrix} \mathbf{R} & \vec{t} \\ 0 & 1 \end{bmatrix} \vec{X}_w \tag{2.17}$$

2.3.3 Normalized Image Coordinates

The normalized image coordinates are very useful to refer to the coordinates in CCF as they make the calculations less confusing and improve the readability. Let $\vec{x}_n$ be the normalized image coordinates, then the 3D point $\vec{P} \overset{def}{=} [X_c, Y_c, Z_c]^T$ w.r.t the CCF can be converted into the normalized camera coordinate $\vec{x}_n$ by diving each coordinate with the depth Z_c.

$$\vec{x}_n = \begin{bmatrix} \frac{X_c}{Z_c} \\ \frac{Y_c}{Z_c} \end{bmatrix} \tag{2.18}$$

Let (x_o, y_o) be the coordinates of the principal point in pixels, f_x and f_y be the focal length of the camera in pixels in the x and y direction respectively, then the pixel coordinates (x_{im}, y_{im}) can also be converted into the normalized coordinate $\vec{x}_n$.

$$\vec{x}_n = \begin{bmatrix} \frac{x_{im} - x_o}{f_x} \\ \frac{y_{im} - y_o}{f_y} \end{bmatrix} \tag{2.19}$$

2.4 Stereo Camera

In the previous section, I talked about the image formation of a pinhole model-based camera. In this section, I will be focusing on a stereo camera setup. A stereo camera is a camera system that consists of two monocular cameras separated by a fixed distance called *baseline* and generates an image pair at any given instant of time. The pixel coordinate can not be remapped back to the world coordinate given only a monocular camera as the depth information is lost during the projection but in a stereo camera, the world point can be accurately measured using *triangulation* or any other method.

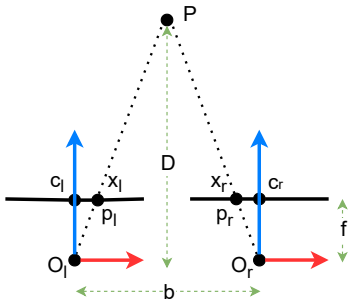


Figure 2.3: A stereo camera system.

2.4.1 Stereo matching for the depth estimation

The stereo cameras are mostly used because of the availability of depth from the stereo images. The conventional way of generating the depth map from the stereo image is to first *rectify* the stereo images such that if an object point can be observed in both images then the pixel coordinate of the object in the vertical axis of the image remains the same for both images and only the pixel coordinate in the horizontal axis differs. If the stereo images are not rectified then their *Epipolar geometry* ([46]) between the two monocular cameras should be known. The purpose of performing the image rectification or estimating the Epipolar geometry is to simplify the process of *stereo matching*.

Stereo matching is an algorithm to find the correspondence of the pixels from the left image to the right image of the stereo pair. In the case of the rectified images, the correspondences shall lie on the same horizontal axis. The stereo-matching algorithm proposed in [24] is one of the most used and efficient stereo-matching algorithms. In this project, I have used the Semi Global Block Matching (SGBM) method from the OpenCV based on [25] to estimate the *disparity* image from the stereo images for the KITTI data. This algorithm is different from the original as it matches the blocks and not individual pixels and it doesn't implement the mutual information cost function.

In the case of the rectified images, the 2D displacement information for the pixels in the left and right images can be encoded into 1D (as the pixel coordinate of the correspondence on the vertical axis of the image is the same), therefore, 1D displacement can be represented by another array of the same size as the original image. This array is called a *disparity map*. The value (also called disparity) at any index in the disparity map tells us the horizontal shift of the pixel in the left image at the same coordinate.

Let b be the baseline of the stereo setup, f be the focal length, (x, y) be the coordinates of the regarded pixel in the image, $d(x, y)$ be the disparity of the pixel (x, y), and $D(x, y)$ be the depth of the pixel (x, y), then the depth $D(x, y)$ can be estimated using the *triangulation* method.

$$D(x, y) = \frac{bf}{d(x, y)} \tag{2.20}$$

Chapter 3

Theoretical Background on Motion Estimation

Contents

In the preceding section, I primarily discussed the fundamental principles encompassing pose, monocular cameras, and stereo cameras, all of which constitute the essential groundwork for 3D Geometry-based Computer Vision (CV). This chapter shifts the focus towards delving into the theoretical underpinnings of *motion estimation*. This encompasses motion estimation within images (referred to as 2D motion estimation) as well as the estimation of 3D motion. The latter encompasses two facets: the estimation of camera motion and the estimation of motion pertaining to 3D objects within the scene. These 3D objects could range from individual points within the 3D space to collections of such points, or even complex 3D structures that can be defined by such constituent 3D points (take, for instance, a 3D cube). For the scope of this project, the concentration remains steadfast on the estimation of 2D motion within images and the 3D motion of the camera itself.

The subsequent sections will introduce an array of concepts inherent to the Ferry-SLAM system. These concepts will subsequently be expounded upon in forthcoming chapters, as required. Among the topics to be covered are Visual Odometry (VO), Optical Flow, Scene Flow, features and keypoints, methods for estimating correspondence, and the ultimate pursuit of pose estimation.

3.1 Image patches, Features, and Keypoints

It is important to talk about the *image patches* first before the *features* and the *keypoints* because only in rare cases, some 2D points in an image are identifiable by themselves. An example of such a point is the crossing point of two thin lines in an image, or the corner point of a rectangular object in an image. Since lines and objects are only rarely represented in perfect sharpness in an image, such mathematical points which are clear concepts in geometry are rarely directly identifiable in images. Therefore, it makes more sense to speak of image patches first, for instance, rectangular sets of pixels.

Image patches, similarity, and dissimilarity metrics: The most important property of an image patch that is useful for motion estimation is that this patch is *well-recognizable* and *unique* in some sense. In order to make the notion of some image patches being 'well recognizable' into an operation concept, we need some *metric*, some measure that expresses how well two image patches match to each other. The Sum of Squared Differences (SSD) is one such metric. A well-recognizable image patch is therefore a patch that is very different from all others in a given image, given a dissimilarity metric such as the SSD.

Keypoints: Keypoints are well-identifiable mathematical points that are associated with an image. They are always associated with an image patch. Keypoints always have a pair of 2D image coordinates, but whether these coordinates are integer-valued or real-valued is a design option. Very often (but not always) keypoints are simply defined as the center of a sufficiently prominent image patch. As they are mathematical points, keypoints live in the domain of geometrical entities and allow for geometric CV algorithms, whereas patches live in the domain of image signals, most often in sampled, thus discrete (non-continuous) images.

Features: According to usual dictionary definitions, a *feature* is a distinctive attribute or aspect of something. In the context of CV, this term is very often (mis)used to express a distinctive part of an image that is often used for establishing point-to-point (most often: 2D-point-to-2D-point) correspondences in motion analysis or used to characterize an object by a set of such distinctive visual parts. In the CV literature, the term 'feature' is very often used for the combination of a *descriptor* computed from some image patch and a keypoint associated with this patch.

Often used feature operators (detectors, keypoints, and descriptors)

A review of different features and keypoints is given in [15] which I have summarised here. The most common types of features and the keypoints used in the V-SLAM are listed below.

- **SIFT**: [31] proposed a Scale Invariant Feature Transform (SIFT) algorithm that can extract the unique keypoints and their descriptors from the image. These features are scale, rotation, and the affine invariant. This feature extractor is relatively slower than other methods because it computes the difference of Gaussians at different pyramid levels of the image.
- **SURF**: The Speeded Up Robust Features (SURF) detector ([3]) is similar to the SIFT feature detector but it uses the *Hessian matrix* to find the features. It is computationally faster than the SIFT feature detector.
- **ORB**: The Oriented Fast and Rotated Brief (ORB) algorithm ([39] combines the speed of the Fast (Features from Accelerated Segment Test) features ([38]) and robustness of the

BRIEF (Binary Robust Independent Elementary Features) descriptors ([9]). The features are rotation invariant but they can be turned into scale invariant with the help of image pyramids.

- **Harris Corners**: The Harris corners ([21]) are the keypoints or the detected corners in the image that have gradient variations in different directions around the regarded pixel.

- **Good Features to Track (GFTT):** The GFTT keypoints ([43]) is the improved version of the Harris corners in terms of the simplicity and robustness. The algorithm computes the minimum Eigenvalue of the structure tensor around the pixel and checks the candidacy of the pixel to be a keypoint. Pixels with high Eignevalues correspond to areas with strong intensity variations and hence, indicate potential keypoints.

3.2 Matching and Correspondences

As mentioned before, the correspondence estimation or the keypoint-to-keypoint matching is one of the steps in the geometry-based VO approach. To estimate the motion between two or more images, we need to find the correspondences of the keypoints between two images. The correspondence problem can be split into two sub-problems. First, we need a search technique that can search for the keypoint or patch from one image in the second image ([16]). Also, we need an *error metric* that can provide a measure to check if the keypoint in the second image is the correspondence of the keypoint in the first image. For example, if we have a patch in the first image and we want to find where this patch is located in the second image, we try to translate the first patch on the second patch and calculate the error for each translation. The translation with the minimum error may be the potential correspondence. This scenario is an example of *translation alignment*. Before diving into different corresponding algorithms, let us look at the error metrics that are most common in the literature and used very frequently by the corresponding estimation algorithms.

3.2.1 Error Metrics

Let i be an index of the pixel coordinate x_i, $I_o(x_i)$ and $I_1(x_i)$ be the template image and the reference image containing the pixel coordinate x_i respectively, $u = (\partial u, \partial v)$ be the shift of the template image $I_o(x_i)$ w.r.t the reference image $I_1(x_i)$, e_i be the residual error for the pixel coordinate x_i, then the residual error e_i can be computed by taking the difference between the shifted template image $I_o(x_i + u)$ and the reference image $I_1(x_i)$.

$$e_i = I_o(x_i + u) - I_t(x_i) \tag{3.1}$$

Let E be an error metric and $\rho(x_i)$ be a function of residual error e_i, then the error metric E can be computed using the following equation.

$$\begin{aligned} E &= \sum_i \rho(I_o(x_i + u) - I_i(x_i)) \\ &= \sum_i \rho(e_i) \end{aligned} \tag{3.2}$$

Sum of Squared Differences (SSD)

It is the summation of the square of the residual error for each corresponding pixel of the two patches.

$$E_{SSD}(u) = \sum_i \rho_{SSD}(e_i) = \sum_i e_i^2 = \sum_i [I_1(x_i + u) - I_o(x_i)]^2 \tag{3.3}$$

Sum of Absolute Differences (SAD) or L_1 norm

It is fast to compute but not differentiable at the origin so not well suited for gradient descent.

$$E_{SAD}(u) = \sum_i \rho_{SAD}(e_i) = \sum_i |e_i| = \sum_i |I_1(x_i + u) - I_o(x_i)| \tag{3.4}$$

Weighted (Windowed) SSD

This metric associates a spatially varying per-pixel weight with each of the two images being matched. It partially or completely down-weight the contributions of certain pixels such as we can ignore pixels that may lie outside the original image boundaries. Let w_o and w_1 be the weighted windows for the template and reference image, then the error metric E can be computed using the weights and the residual error e_i.

$$E_{WSSD}(u) = \sum_i w_o(x_i) w_1(x_i + u)[I_1(x_i + u) - I_o(x_i)]^2 = \sum_i w_o(x_i) w_1(x_i + u) e_i^2 \tag{3.5}$$

Root Mean Square

If a large range of potential motions is allowed, WSSD (Weighted SSD) can have a bias towards smaller overlap solutions. To counteract this, the WSSD score can be divided by overlap area to compute per-pixel (mean) squared pixel error.

$$E_{RMS} = \sqrt{E_{WSSD}/A} \tag{3.6}$$

where area A is

$$A = \sum_i w_o(x_i) w_1(x_i + u) \tag{3.7}$$

3.2.2 Matching algorithms

There are different algorithms that can be used to find the correspondences of the keypoints from one image to another. These algorithms can be categorized into three categories: (1) Block Matching (BM), (2) phase correlation (PhC) based matching, and (3) Differential matching.

Block matching

In Block Matching (BM), the basic idea is to compare a region (or block) of pixels in one image (called the reference image) with corresponding regions in another image (called the target image) to find the best match ([16]). The "best match" is typically determined by minimizing an error metric, such as the SSD, between the reference block and candidate blocks in the target image.

Phase correlation-based matching

The phase correlation (PhC)-based correspondence estimation is a powerful technique that leverages frequency domain information to accurately estimate motion or displacement between images ([34]). It relies on the Fourier Transform and exploits the phase information of the frequency domain representation of images. It is less sensitive to changes in lighting, contrast, and noise compared to direct pixel-based methods.

Differential matching

The most popular differential matcher is the Lukas-Kanade (LK) based differential tracker proposed in [32]. The principle of the LK matcher is to select a rectangular window (usually around a keypoint) from an image (say image A) and find the corresponding window in another image (say image B). Instead of searching the reference window from image A in the whole of image B as in template matching, it restricts the search to a predefined search area. The search area is defined by its center which is the initial best guess of the correspondence window and the bounds of the search area (length and width in case of rectangular search area). Both methods BM and LK use a loss function and compute the gradient of this loss function to determine the direction in which the corresponding window can slide but unlike the BM method that slides the corresponding window in integer pixel increments, the LK matcher moves the corresponding window in fractional pixel increments and because of this reason, it is put in the category of differential matching algorithms. The search stops when the stopping criterion satisfied which is minimizing a given loss function, for example, SSD, or the maximum number of iterations. In the LK, the search window size is much smaller than the BM but it is able to provide a resolution of fractions of a pixel. It is reasonable to assume that LK matcher provides a resolution of $1/10^{th}$ of a pixel.

3.3 Optical flow and scene flow

Optical flow [4] is another CV technique that represents the pixel-level motion of the objects between consecutive image frames from the camera. In simple words, it determines the 2D motion of pixels from one frame to another. The optical flow is based on the *Brightness Constancy Constraint* which implies that the intensity of the corresponding pixel values remains the same in the two images. Optical flow provides a lot of information about the scene and the camera and this technique can be used in ego-motion analysis, object tracking, and structure from motion.

Scene flow is similar to the optical flow but it measures the 3D motion of the pixels. In the presence of the depth map, the pixel coordinates can be converted into 3D Cartesian coordinates, and the estimation of the change of the 3D coordinates of the pixel from one frame to another is called scene flow. The scene flow can be computed from the 3D point clouds directly as in [36]. They constrained the scene flow using graph *Laplacian* such that the points in a local region moved rigidly and all the points in the source point cloud moved "as-rigid-as-possible".

3.4 Perspective-n-Point (PnP) based motion estimation

PnP[1] is a technique used to measure the motion between two frames using the correspondences of the keypoints. It first projects the keypoint from the image space to the 3D space and

[1] OpenCV library implements the PnP problem in different ways and the link to the tutorial is here.

then reprojects the 3D on the other image. Finally, it optimizes the motion by minimizing the *reprojection error* ([17]).

3.5 Ego-motion Estimation and Visual Odometry

Ego-motion estimation, or self-motion estimation, refers to the self-motion of the camera or vehicle, as opposed to the motion of other objects in the scene. The VO is a technique in the CV and robotics to estimate the motion of the camera by analyzing the images from the camera as it moves in the scene. The objective of the VO is to estimate the pose of the camera build a trajectory and create a 3D scene using the pose information if needed. The main principle behind the VO is to measure the 2D displacement of the pixels (also referred to as *Optical Flow*) in the image and then compute the pose change using the optical flow. The VO used to be only geometry based but since the last decade, the learning-based VO has also emerged.

3.5.1 Geometry-based Visual Odometry

In the geometry-based VO, we extract unique *features* or *keypoints* (see section 3.1 for the difference between the keypoints and the features) and establish the *correspondences* between them. Based on these correspondences the relative camera motion is estimated. Two main categories of the geometry-based VO are: (a) feature-based or indirect methods and (b) appearance-based or direct methods. In the features-based approach, a few features from the whole image are selected and matched between the images, and in the appearance-based approach, the pixel intensities of the image are considered. Both of these methods can be further categorized into dense and sparse methods. In the dense method, all the pixels in the image are used, whereas, in the sparse method, only a smaller set of pixels are used. The keypoints-based method comes under the category of the sparse-indirect method as it uses selected pixels from the images and matches them by minimizing some error metric.

3.5.2 Learning-based Visual Odometry

Learning-based VO is an emerging technique that uses deep neural networks to learn the patterns from the visual data to estimate the camera motion and doesn't rely on geometric calculations as in the geometry-based methods ([30]).

3.6 Keyframing

Keyframing refers to the process of selecting specific frames or time instances from a sequence of sensor data (such as images or sensor readings) that are used as reference points for estimating the motion or movement of a camera or vehicle (ego-motion). These selected frames are called keyframes. Keyframing is used to simplify and optimize the computation of ego-motion. Instead of processing every single frame in a sequence, which could be computationally expensive and unnecessary, keyframes are strategically chosen frames that represent significant changes in the environment or camera motion.

3.7 Independent Moving Objects (IMOs)

Any kind of VO or V-SLAM system is always designed for a mobile platform that can move in a constrained or in open space environment. Irrespective of the operating environment, there are always some scenarios in which there could be another potential object that the ego-vehicle should avoid at all costs. If these objects are moving then they are referred as IMOs. It is important to identify the moving objects before the estimation of the ego-motion to avoid the computation of the relative motion between camera and the moving objects. If the keypoints fall on such moving objects, then the optical/scene flow generated from these keypoints will create a bias in the ego-motion estimation.

CHAPTER 4

Datasets and Sequence Acquisition

Contents

It is very important to validate the ego-motion system by comparing the results with the ground truth. This is where datasets come in handy as they provide the data from real scenarios with ground truth information. On these data, I can run my system and check how much the results deviate from reality or the ground truth. The ground truth available from these data sets comes from very precise calibration and testing. It is quite possible that this ground truth is not the actual truth ([6]) but it is not far from it either.

4.1 KITTI Dataset

KITTI is one of the most popular datasets for 3D scene flow estimation ([19]) with an application for mobile robotics and autonomous driving. I have used the KITTI dataset and tested the system on different KITTI sequences. There are 22 sequences available in the KITTI dataset; the following information is provided for each sequence.

- **Images**: The dataset provides the color and grayscale rectified stereo images.
- **Timestamps**: The relative timestamps are also provided for each corresponding frame in the sequence. The timestamp for the first stereo frame is 0.0.
- **Camera Params**: The projection matrix for all the cameras with respect to the left camera.

Apart from the above information, the absolute pose for each timestamp is also given for 11 sequences out of the 22 sequences. This pose is relative with respect to the first frame and measured in the left Camera Coordinate Frame (CCF).

Table 4.1: Special cases from the KITTI dataset

Seq.	Start frame	End frame	Feature
00	0	86	straight line motion with moving objects
00	87	130	turning around the corner
00	220	400	straight line motion without moving objects
00	530	560	stationary camera
00	4000	4350	curve with trees on right side
01	220	860	highway with moving cars and distant buildings
02	1060	1290	vehicle following on street
05	2330	2390	waiting while cars are crossing
06	0	900	loop closure
12	500	560	highway without vehicles and only trees
14	110	510	square loop closure

4.1.1 Case Scenarios

The robustness of any system can be tested by checking its performance in the edge cases of its applications. These edge cases are those scenarios in which the system is very vulnerable to failures and therefore, these cases should be more focused. The potential edge cases from the KITTI dataset are highlighted in table 4.1. It is important for the system to handle all these cases without running into a state of failure.

4.2 Dataset Generation using ZED Camera

The KITTI dataset has a huge diversity in the sequences but they are not recorded from the maritime context. There are no water and fellow boats in any of the KITTI sequences at all which are very crucial for the testing of the proposed system, therefore, I also created some sequences using the ZED camera (see chapter D, p.183). These sequences are recorded in the canal of Trondheim, Norway. These sequences are generated using three different setups recorded on two different days[1]. Unlike the KITTI dataset, the ground truth is not available for these sequences. These setups are discussed in the following sections.

[1]The description of the recorded dataset can be obtained using this link.

Table 4.2: Setup settings for different sequences

Sequence	Setup Settings			
	Date	Resolution	FPS	Compression Type
Z1-FT-MB-SEQ-1	14-03-2023	2208x1242	15	H264-LOSSY
Z1-FT-MB-SEQ-2	14-03-2023	2208x1242	15	H264-LOSSY
Z1-HH-MB-SEQ-1	14-03-2023	2208x1242	15	H264-LOSSY
Z1-HH-MB-SEQ-2	14-03-2023	2208x1242	15	H264-LOSSY
Z2-FM-EB-SEQ-1	10-05-2023	1920x1080	15	LOSELESS
Z2-FM-EB-SEQ-2	10-05-2023	1920x1080	15	LOSELESS

4.2.1 Setup Z1-FT-MB

In this setup, a single ZED camera model is taped to the top of a small motor boat and the sequences were recorded using the ZED Explorer at an FPS of 15 with a resolution of 2208x1242 using H264-LOSSY compression. In this setup, different kinds of sequences were recorded. In sequence Z1-FT-MB-SEQ-1, the boat drove towards the bridge, took a 180-degree turn, and crossed from where it started. In the other sequence Z1-FT-MB-SEQ-2, the boat was moving slowly and the camera was facing the sun directly so the exposure was reduced for the recording.

4.2.2 Setup Z1-HH-MB

In this setup, a single ZED camera model was hand-held and faced toward the back of the boat. The sequences recorded in this setup were Z1-HH-MB-SEQ-1, Z1-HH-MB-SEQ-2, etc. In Z1-HH-MB-SEQ-1, the boat was moving away from the shore at high speed. Because of the high waves and high speed of the motorboat, the camera was very unstable and there were a lot of jerks in the sequence. However, in Z1-HH-MB-SEQ-2, the boat was entering the Fjord from the open waters calmly.

4.2.3 Setup: Z2-FM-EB

Unlike the previous two setups, two ZED cameras were used to record the sequences. One ZED 1 camera was mounted on the port-bow side of the Milliampere-2. and the other ZED camera (ZED 2) was mounted on the starboard bow of the boat. Both of the cameras were front looking with a lateral displacement of approximately 1.89 meters (see fig. 4.1). In one of the sequences Z2-FM-EB-SEQ-1, the milliampere-2 is docking, and in the other sequence Z2-FM-EB-SEQ-2, the milliampere-2 is docked and maintaining its position with the help of the dynamic positioning system. In sequence Z2-FM-EB-SEQ-2, the cameras are looking towards the opposite side of the dock and another motorboat comes into the view of the camera and performed some maneuvers.

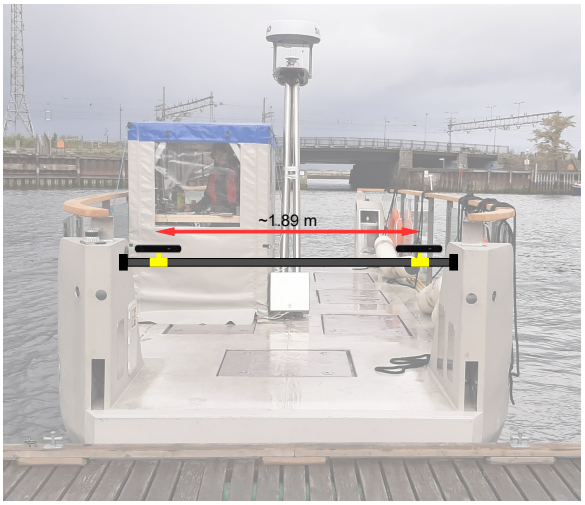
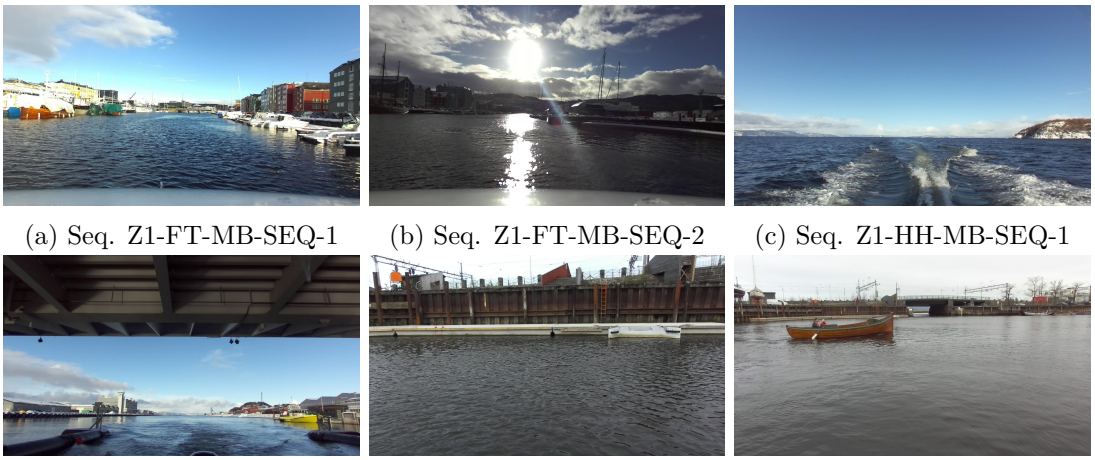


Figure 4.1: Z2-FM-EB setup on MilliAmpere-2. Graphics are overlaid on images taken from [7].



(a) Seq. Z1-FT-MB-SEQ-1 (b) Seq. Z1-FT-MB-SEQ-2 (c) Seq. Z1-HH-MB-SEQ-1

(d) Seq. Z1-HH-MB-SEQ-2 (e) Seq. Z2-FM-EB-SEQ-1 (f) Seq. Z2-FM-EB-SEQ-2

Figure 4.2: The images taken from the left camera of the ZED from the different sequences.

Part III

The **Ferry-SLAM** System and its Modules and Components

CHAPTER 5

THE ARCHITECTURE OF THE Ferry-SLAM SYSTEM

Contents

Ferry-SLAM is a system designed to address three major problems related to the Computer Vision (CV) for maritime applications: (1) *3D Scene Analysis*, (2) Object Detection, and (3) Ego-motion Estimation.

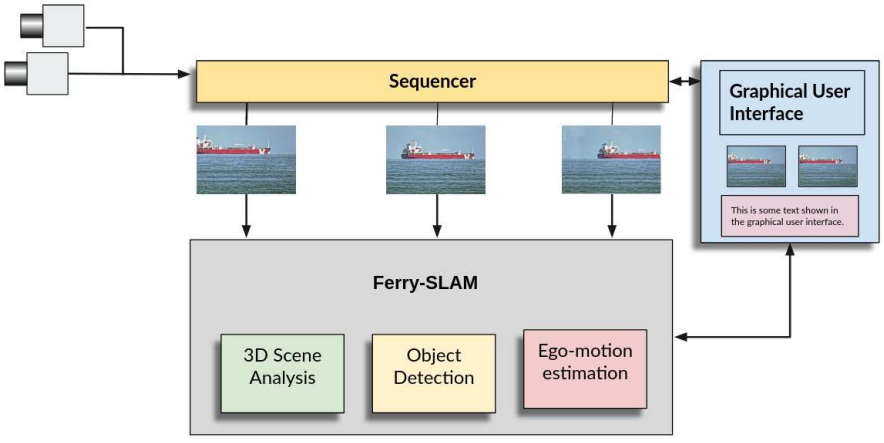


Figure 5.1: The overall architecture of the Ferry-SLAM system.

Figure 5.1 shows the main modules of the Ferry-SLAM and shows how the input stereo data flows from the *sequencer* through the Ferry-SLAM and finally visualized by the *Graphical User Interface (GUI)*. The sequencer and the GUI are the helping tools that are separated from the main Ferry-SLAM system.

As mentioned before, the Ferry-SLAM system consists of 3 sub-systems that can run together or independently. Each sub-system will be explained in detail in the later chapters but a brief review is given in the following sections.

5.1 3D scene analysis

This module consists of building a 3D representation of the scene, which is a combination of both geometry and semantics such that the 3D representation can be used for situation awareness. In the present work, this 3D scene analysis is done on the basis of a single stereo pair but in the future, it can be extended to explicitly use the temporal continuity of the scene, i.e., exploitation of the *temporal correlations*.

The 3D scene analysis consists of transforming the point cloud into a set of *surfaces* using Machine Learning based semantic segmentation networks and the plane fitting using geometry-based methods (see chapter 7, p.39). One of the semantic networks is *Aquanet* which can segment the water plane and also the other objects such as boats, buildings, etc., and the other network is *YOLO* which is used to segment the objects (boats, persons, etc.) from the environment. From different surfaces, the *navigable area* (the water surface in the case of maritime applications) can be detected which is of the most interest for the navigation of the maritime vehicle.

Besides the above-mentioned methods to find the surfaces, there is another approach that can be used to find the horizontal and upright surfaces using the disparity data only (see chapter

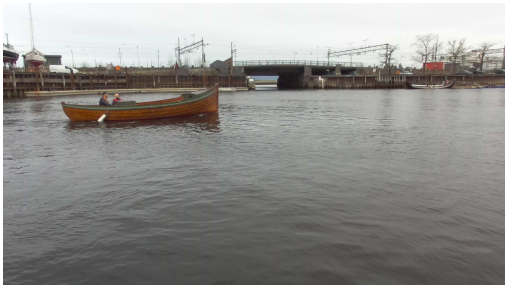


Figure 5.2: Input color image

Figure 5.3: Semantic labeling of the pixels

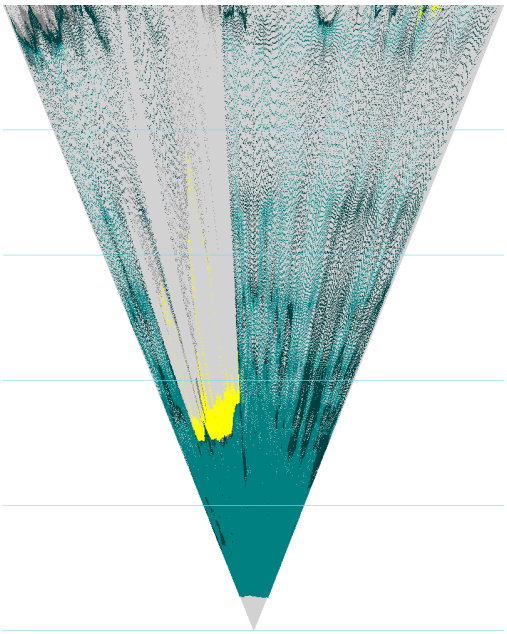


Figure 5.4: Depth's BEV

8, p.55). The visualization of the scene is done by the projection of the selected 3D data on a horizontal plane also referred to as the *Bird's Eye View (BEV)*. In BEV, the 3D scene is looked at from the top such that only the horizontal surfaces and the boundaries of the object appear (see chapter 9, p.63). BEV is an important element of the 3D Scene Analysis sub-system as it provides both an intuitively understandable visual representation as well as provides a map-like structure that can be evolved into a real local map. For the BEV, it is possible to base it on the semantic labels, as well as on the upright/horizontal pixel classification. For instance, in a semantic-colored BEV map (see figure 5.4), all the holes can be filled and the uncertain areas can be correctly flagged, but I did not work on this due to the lack of time but it can be addressed in future work.

5.2 Object Detection

There are different kinds of obstacles that a maritime vessel has to avoid during the operation. These obstacles can either be static (walls, docks, etc) or dynamic/moving (boats, persons, etc). I will use the term '*obstacle detection* for the static obstacles and the term 'object detection' for the potentially moving obstacles. As the methods used in this project work on single stereo pairs, decisions on whether an object actually moves cannot be made, and I defer such research to future work.

This module also uses the results from the semantic networks (*Aquanet* and *YOLO*) for the identification of all the potential moving obstacles and fuses the semantic labels with the geometric indicators (whether the surface is upright or horizontal) to detect the static obstacles. This module is lesser developed than the previous module (3D scene analysis) due to the limited time.

The module uses Machine Learning and geometry-based techniques to solve this problem. It accepts a color image and the corresponding depth data and creates a labeled mask that labels the water (free zone) and the fellow boats (see chapter 7, p.39). Every point in the mask that is not labeled as water is a potential object and shall be avoided.

5.3 Ego-motion estimation

This sub-system is responsible for the estimation of the ego-motion using the keypoints-based pose estimation approach (see section 3.5.1, p.22). In this approach, the keypoints are detected in one image and their correspondences are found in the other image, and using these correspondences, the ego-motion is estimated. The architecture of this sub-system has been discussed in chapter 10.

Chapter 6

Experimental Framework used in the Ferry-SLAM project

Contents

The handling of the data coming from the camera or the dataset and the visualization of the intermediate results is as important as developing the system as it simplifies the debugging process and can help to identify the edge cases where the system fails. The intermediate results could be images, graphs, or hyperparameter values. The system for Visual Odometry (VO) or Visual SLAM (V-SLAM) is always tested against some dataset and it becomes necessary to have direct control over the dataset while processing it instead of directly feeding the data to the pipeline. In this chapter, I will talk about a framework that handles the dataset as well as the visualization of the intermediate results. It consists of two modules: (1) a sequencer and (2) a Graphical User Interface (GUI). The former handles the dataset while the latter is responsible for the interactive communication between the sequencer and the Ferry-SLAM system and also handles the visualization. Figure 5.1 shows how the sequencer reads the images from the stereo camera or the stereo images dataset and passes it to the Ferry-SLAM and the GUI interacts with the sequencer and visualizes the intermediate results.

6.1 Sequencer: A dataset handling tool

Sequencer is a concept developed at NTNU under the supervision of Professor Rudolf Mester to handle the dataset more efficiently w.r.t the debugging of the system and to overcome the limitations of the traditional way of dataset handling. In this section, I will first talk about the classical way to handle the dataset followed by the original idea of the sequencer and my contributions to it.

6.1.1 The traditional way of dataset handling

In an image-based dataset, the traditional approach is to provide the file path in the code, load the images one by one, or load all the images in the buffer, and process them sequentially. The intermediate results are visualized using some independent libraries only. This approach is really fast and simple to implement but it becomes useless when the system breaks in the middle of the dataset while handling some edge cases. The only way to reproduce the outcome of the system is to run it again until it reaches that point again or change the code directly, to begin with the problematic frame in the dataset. Both of these solutions are very inefficient and consume a lot of time while debugging. Also, the need to jump between the frames arises during the tuning of the hyperparameters of the system which can not be done dynamically.

6.1.2 Sequencer 1.0: Command-based dataset handler

The original sequencer (a tool to handle the dataset and visualize the results) was developed by one of the previous students supervised by Prof. Rudolf Mester. This tool was a command-based application. The user can enter the path to the dataset in the terminal and can jump to the next frame, previous frame, or any arbitrary frame using the terminal. It also provides the functionality to convert datasets in video format to images. This sequencer is very fast because of low-level data handling and visualization using the core libraries, however, it was restricted to the terminal only. The functionalities were also limited to jumping between frames only. In this project, I needed more control over the dataset for faster debugging and a user-friendly interface. This motivation led me to develop the next version of the sequencer called Sequencer 2.0.

6.1.3 Sequencer 2.0

Sequencer 2.0 (also referred to as sequencer in this project) offers more features than sequencer 1.0 in terms of the types of the dataset as well as the control over the frames which are mentioned below.

Acceptance of the ZED dataset

The data recorded using the ZED camera is always saved in its own dataset format and to access the frames, their functions from their SDK have to be used. The sequencer provides a wrapper for the SDK such that the user can access the ZED dataset without looking at the SDK API.

Accessing frames by filename and frame number

As with any standard image dataset, it is expected that the file names of the image should have a certain pattern. For example, the images in the *KITTI* dataset have the *"00XXXX.png"* pattern in their names, where the *"XXXX"* is the frame number. The sequencer expects the same or similar pattern in the file names where the maximum characters are constant (6 in the case of the KITTI dataset) such that the sequencer can run the files in arbitrary frame jumps (see section 6.1.3), including in reverse order.

Handling more than the image frames

The images are not sufficient for the Ferry-SLAM system as it also needs the intrinsic and the extrinsic calibration parameters of the camera (see section 2.3, p.12), the timestamps of the

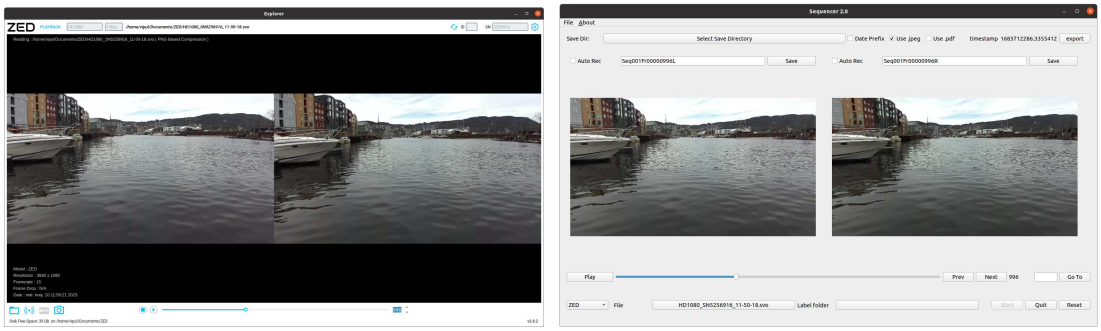
frames, the pose data for the VO if available. The sequencer accepts this data and store it in the buffer and returns it as the meta-data corresponding to the particular frame.

Accept control files to jump between frames

As mentioned before, the sequencer needs the filenames to have a certain pattern. This feature is also used by the *control files*. The control files are the *json* files that contain the interested subsequences in the dataset. These subsequences contain the start frame number and the ending frame number. When the sequencer reads the control file, it parses the first subsequence jumps to the starting frame number of the subsequence, and provides the images until the ending frame number. Once, it reaches the end, it jumps to the second subsequence if available and keeps repeating the steps until the whole file is parsed.

6.2 Graphical User Interface (GUI)

The motivation to develop a GUI similar to the *ZED Explorer* (see Fig. 6.1a) that can provide multiple image windows simultaneously and a controller to jump between the frames easily. The ZED Explorer is a proprietary application from the Stereo-Labs and can not be interfaced with the Computer Vision (CV) application. It is limited to only visualization of the dataset generated using the ZED camera. The GUI is similar to the ZED Explorer but it can be interfaced with the CV applications for the visualization. it also provides an interactive interface for the sequencer and other helpful tools to save the intermediate results and much more. The features and the usage of the GUI are given in appendix E.



(a) GUI for the ZED Explorer. (b) GUI for the sequencer

Figure 6.1: Comparison between the GUI of the ZED Explorer and the sequencer.

CHAPTER 7

Semantic Segmentation and Water Surface Recognition

Contents

In the keypoints-based computer vision pipeline, avoiding the keypoints on the sea surface and the sky is always preferred for the ego-motion of a maritime vessel. The sea is very dynamic and keypoints detected on the sea can not be tracked in the next image. Therefore, all kinds of temporal appearance-based tracking cannot work with such keypoints, and the sky, it is a textureless region without any depth estimates. In some scenarios, the presence of clouds can be very useful for estimating the rotation as the clouds come under the category of the far region, however, they are textureless as well, and detecting and tracking the keypoints accurately could be another problem. Therefore, the sky can be neglected but neglecting the sea completely is not a wise solution as it still can be useful to extract other information.

We want to find the orientation of the camera and if we have the water plane, we can also estimate the orientation of the ego-vehicle relative to this water surface and track this orientation while the ego-ship is moving. As the water surface is characterized by waves, this orientation is not exactly constant over a period of time. Furthermore, we need to know the orientation w.r.t. the water surface since the water surface is the reference plane for producing different kinds of 2D maps such as the Bird's Eye View (BEV). Also, if the sea surface is known then the objects (potentially obstacles) floating on it can be extracted. Therefore, the sea surface should be handled differently.

Taking these advantages into consideration, I decided to develop strategies that can identify the water plane. I explored two different types of approaches to solving this problem. My

first approach is based on the current advancements in the field of Artificial Intelligence (AI). Machine learning, a sub-field of the AI, has emerged as a potential solution to solve problems more human likely. Machine learning algorithms are usually trained on huge data and designed to solve a particular problem. They are mostly considered black boxes because of their lack of explainability. In the Computer Vision (CV) applications, these machine learning algorithms usually use the Convolutional Neural Network (NN) ([35]) or the vision-transformers ([12]). The transformers were mostly have been used in Natural Language Processing related tasks, but in 2020, they were introduced to solve the vision problems, however, the Convolutional NN is still dominating in the Computer Vision (CV) applications. In maritime applications, the usage of machine learning is still limited because of the limited dataset, therefore, my second approach is based on the conventional geometry-based methods to compensate for the lack of advancements in this domain.

In this chapter, I will first talk about the sea segmentation using Convolution NN and then I will discuss the Random sample consensus (RANSAC) based plane segmentation algorithm. The identification of the potential objects on the water is very crucial for collision avoidance and object tracking, therefore, I will also use another Convolution NN called YOLO-v8 to detect the boats on the water. Finally, I will discuss the fusion of the results from the different proposed segmentation algorithms.

7.1 Sea segmentation using Aquanet

In the automotive industry, segmenting the surface on which the vehicle can move is usually referred to as free space detection problem. For the maritime vessels, this free surface will be the sea, and segmenting it from the rest of the image is one of the requirements of this project. [40] proposed a self-supervision-based Fully Convolutional Network (FCN) to segment out the free space i.e. the roads. For the sea-segmentation, a lot of work already has been done in sea-land segmentation using the remote sensing images [29], [42], and [11], however, from the V-SLAM's application point of view, the network should be able to differentiate the sea based on the frontal images rather than the top images. I found one Convolution NN called Aquanet ([14]) that was able to fulfill the requirements of this project which are the ability to segment the sea using the frontal images as well as the public availability of the network.

7.1.1 Implementation of the Aquanet

[14] proposed a Convolutional NN based network called Aquanet to label each pixel into *num_classes* different classes in which *num_water_classes* belong to the waterbodies and *num_other_classes* classes belong to the general bodies such as a person, car, sky etc. This network is trained on the Atlantis dataset released by the authors in the same paper. I decided to go with this network because of two reasons.

- **Open-source:** The network architecture is already provided by the authors along with the trained weights.
- **Focused on Maritime:** The labels of the dataset are focused on maritime applications. The main labels that I was looking for (sea, sky, and boat) were already present in the dataset and the network can identify them as well.

The implementation of the Aquanet is shown in Fig. 7.1 and represented in algorithm 1. The Aquanet's NN takes a *network_size* size of the color image as an input and outputs a probability

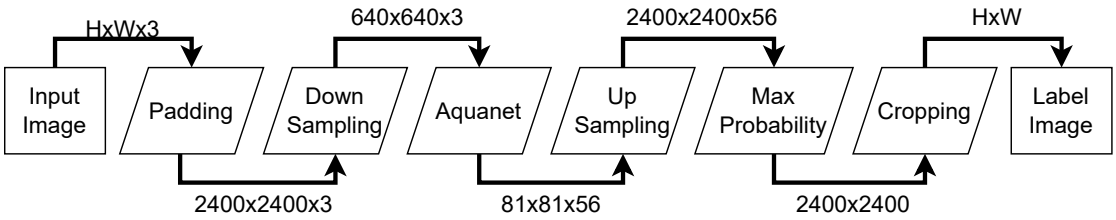


Figure 7.1: Implementation of Aquanet. The input image is a color image with dimension $H \times W$ and the output is a single channel label image with the same dimensions.

map corresponding to each label. To use this network in the algorithm, the image was first padded with zeros (step 3) on the border until the width and height of the image reaches *padding_size* pixels. Then, the padded image is downsampled (step 4) to the network's input size that is *network_size* using Pytorch's implementation of the interpolation function. The interpolation uses the *bi-linear* interpolation method with aligned corners. The processed image is passed to the network (step 5) and the probability map was stored. In the source code of Aquanet, the authors were using the *Upsample* function of Pytorch to upsample the probability map to the same size as the input padded image. This function is deprecated now and it is replaced with the same interpolate function (step 6) with the same settings. This upsampled probability map is then reduced to a label image by considering the label with the maximum probability (step 7). It results in a padded label image which was cropped to match the original input size (step 8).

7.1.2 Results and discussion on Aquanet

The processing time to get the label image from the input image is noted to be ≈ 1.8 seconds when an *input_size* image is used on an NVIDIA GeForce RTX 3050 Laptop GPU. Figure 7.2 shows the input and the output of the segmentation pipeline mentioned above. The labeled image (see Fig. 7.2b) is post-processed. The water-bodies labels are replaced with a single label

Algorithm 1 Integration of Aquanet in the pipeline

```
1  procedure GET_SEMANTIC_LABELS(img)
2      model ← load_model()
3      padded_img ← pad(img, padding_size)
4      downsampled_img ← interpolate(img, network_size)
5      predictions ← model(downsampled_img)
6      upsampled_img ← interpolate(predictions, padding_size)
7      label_img ← argmax(upsampled_img)
8      label_img ← remove_pad(label_img)
9      label_img ← merge_labels(label_img)
10     return label_img
```

Table 7.1: Parameters for the Aquanet

Parameter	code name	Value
Total number of classes	num_classes	56
Number of water classes	num_water_classes	35
Number of other classes	num_other_classes	21
Padding size	padding_size	2400
Input image size	input_size	1080×1920
Network's input size	network_size	640×640

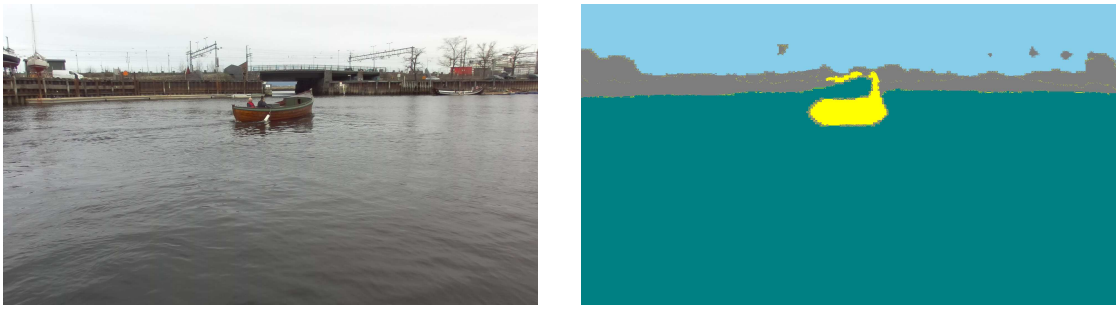
(step 9) and colored in teel, the sky is colored in blue, boats/ships are colored in yellow and the rest of the labels are colored in gray. It was observed that the Aquanet manages to label water bodies correctly most of the time but it fails when little or no water is present. Also, the labeling of boats and ships is not accurate as seen in Fig. 7.2b. The network doesn't give pixel-level accuracy but it can give a rough estimate of the different types of objects present.

In conclusion, Aquanet is relevant but not a reliable semantic segmentation network for this application. Any good and stable network that has the ability to segment the water surface and the bodies floating on it such as ships, boats, etc., can be used instead.

7.2 RANSAC based water segmentation

The ferries are mostly operated in those water bodies where high waves are very uncommon and the water surface is mostly stable. This prior knowledge of the environment can be exploited to test geometry-based methods. The geometry-based methods can fit a plane to the water surface if a sufficient amount of 3D points that lie on the water surface are given. These 3D points usually come from the LiDARs or the stereo camera.

[28] uses a RANSAC method (like I also do): In this paper, the authors select three points randomly from the point cloud and calculate the plane parameters and subsequently try to enlarge the plane given some thresholds. This method requires about 10 times more computation time than the constrained *Randomised Hough Transform* (RHT) method ([27]) that can find the



(a) Input to the segmentation pipeline. (b) Colored label image

Figure 7.2: Input and output of the Aquanet segmentation pipeline. Image taken from Z2-FM-EB-SEQ-2 sequence is used as an input and the output label image is colored according to the color coding defined in table 7.2.

Table 7.2: Color coding of the labels

label name	label color	color code (BGR)
definitely sea	Teal	(128, 128, 0)
maybe sea	Dark teal	(64, 64, 0)
boat	Yellow	(0, 255, 255)
sky	Blue	(235, 206, 135)
Other identified objects	Gray	(127, 127, 127)
Unidentified objects	Black	(0, 0, 0)

ground plane by applying a coarse prior assumption on the ground plane. In their approach, they first computed the local surfaces from the 3D point cloud such that each local surface fits a central point as well as *K nearest neighbors* to that central point. The large the number of neighboring points, the larger will be the local surface. Finally, they computed the normal distance of these local surfaces w.r.t the Camera Coordinate Frame (CCF) and compared them with the nominal or known ground plane parameters.

The only limitation of their work in my application was the prior knowledge of the plane parameters. In this project, I didn't measure the orientation and height of the camera with respect to the water plane while recording the dataset and therefore, I can not use the constrained RHT-based approach directly which requires approximate plane parameters. I decided to use the RANSAC based approach that can find the water plane with minimal prior knowledge i.e. the approximate location of the water in the 3D point cloud or in the image. I could have used the RANSAC to find the plane parameters first and then use them with the RHT approach but I didn't do that to keep the plane estimation algorithm simple.

In this section, I will talk about the classical RANSAC based plane segmentation and its shortcomings, and in the next section, I will propose my implementation for the plane segmentation that will be more suitable for maritime applications.

7.2.1 Plane segmentation using RANSAC

The RANSAC based algorithms are very robust to the outliers and able to find the geometry given the number of inliers is greater than the number of outliers. To fit a water plane in an open water environment using a raw 3D point cloud. only is not sufficient because of the poor depth estimates of the far points. To overcome such problems, a large number of iterations or the predefined boundary of the water plane will be required. The former will consume a lot of computational time and the latter will require the knowledge of the water plane which we don't have. Instead of fitting the whole point cloud, reducing the number of points in the point cloud by cropping it beforehand can reduce the number of outliers. Therefore, the RANSAC should be able to find the ground/water plane within a few iterations. The three main steps (see Fig. 7.3) to identify the water plane parameters and to label the pixels in the image that belongs to the water plane are discussed in detail in the following sections.

7.2.2 Identification of the plane horizon and cropping of the point cloud

I decided to use the prior knowledge of the *horizon* of the water plane to crop the raw point cloud. The horizon of the water plane can be seen as a line in the image such that the water

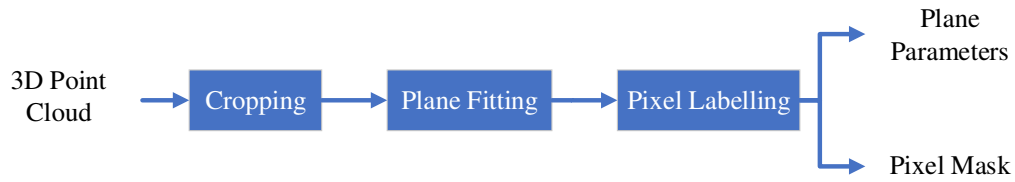


Figure 7.3: Flow chart for the RANSAC based water plane fitting.

Table 7.3: Parameters for the plane segmentor

Parameter	code name	Value
Number of samples	ransac_n	3
Maximum number of iterations	max_itr	5
Distance threshold	dist_th	0.1 meters
Distance rejection threshold	height_th	0.1 meters
Angle rejection threshold	angle_th	5 degrees

plane can not exist above this horizon line, also referred to as the vanishing line in 3D geometry ([22], p.216). It should be noted that the accuracy of the horizon line is not very important as the whole purpose of cropping the raw point cloud is to reduce the number of outliers for the RANSAC. If I do not know the orientation of the camera w.r.t. the water plane which is also happened to be in this project and required to estimate the horizon line, I can assume the pitch to be so that the resulting horizon line is safely below the true horizon line. In other words: the camera looks slightly up. The result of this assumption will be a reduction in the outliers.

As mentioned before, most of the water region will be at the bottom of the image and below the horizon line, therefore, assuming the camera is parallel to the water plane (pitch is zero), I decided to crop out the points that lie above the horizon line and use the lower half of the image for the identification of the plane such that all the points from the point cloud that lie above the xz plane of the CCF are removed. It should be noted that this assumption of pitch is valid only for this project and should be tuned depending on the dataset or the environment.

7.2.3 Plane fitting and identification of the plane parameters

I decided to use Open3D's implementation for the plane segmentation[1] ([47]). The parameters used by this function are given in the table 7.3. Let a, b, c, and d be the plane parameters then the output of the plane fitting algorithm is the plane equation in the CCF.

$$ax + by + cz + d = 0 \tag{7.1}$$

Let $\vec{n}$ be the unit normal vector of the plane and $d_\perp$ be the perpendicular distance between the origin of the CCF and the fitted plane then the plane parameters can be used to compute

[1]The tutorial for this function is given here

the normal vector $\vec{n}$ and the perpendicular distance $d_\perp$.

$$\begin{aligned} \vec{n} &= \begin{bmatrix} \vec{n}_x \\ \vec{n}_y \\ \vec{n}_z \end{bmatrix} \\ &= \frac{1}{\sqrt{a^2+b^2+c^2}} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \\ d_\perp &= \frac{d}{\sqrt{a^2+b^2+c^2}} \end{aligned} \tag{7.2}$$

7.2.4 Pitch estimation from the plane parameters

The identification of the plane w.r.t. the camera can also help us to identify the pitch and the roll of the camera w.r.t. the identified plane. As a reminder, the $z-axis$ and $y-axis$ of the CCF are pointing forwards and downwards respectively. First, the normal vector of the plane ($\vec{n}$) will be projected on the yz plane of the CCF (see Fig. 7.4).

$$\vec{n}_{yz} = \begin{bmatrix} 0 \\ \vec{n}_y \\ \vec{n}_z \end{bmatrix} \tag{7.3}$$

Let $\vec{u}_z$ be the unit vector in the z-direction of the CCF and α be the angle between the $\vec{u}_z$ and the $\vec{n}_{yz}$, then the angle α can be computed by taking the inverse of the cosine of the normalized projection of the $\vec{n}_{yz}$ on the $\vec{u}_z$.

$$\alpha = \arccos\left(\frac{\vec{n}_{yz} \cdot \vec{u}_z}{\|\vec{n}_{yz}\|\|\vec{u}_z\|}\right) \tag{7.4}$$

Let θ_p be the pitch angle of the plane w.r.t. the CCF, then it can be computed from the angle α.

$$\theta_p = \frac{\pi}{2} - \alpha \tag{7.5}$$

It should be noted that the pitch angle θ_p could have been computed by taking the projection of the $\vec{n}_{yz}$ on the unit vector in the y-direction of the CCF directly but I choose the unit vector in z-direction because the Numpy's implementation for the inverse of the cosine[2] returns the angle between $[0, 180°]$ and the pitch angle varies between $[-90°, 90°]$. Therefore, to correct this mapping, I calculated the angle w.r.t. the unit vector in z-direction $\vec{u}_z$ and subtracted it from the $90°$ such that the final pitch angle θ_p varies between $[-90°, 90°]$. Finally, the pitch angle θ of the camera w.r.t. the plane is computed by taking the inverse of the rotation.

$$\begin{aligned} \theta &= -\theta_p \\ &= \alpha - \frac{\pi}{2} \end{aligned} \tag{7.6}$$

[2]The API reference for the NUMPY function to compute the inverse of the cosine is given here.

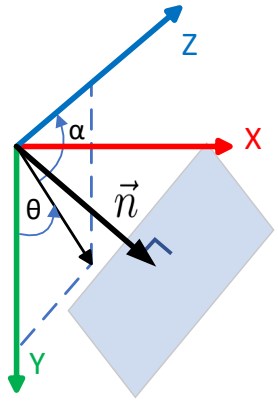


Figure 7.4: Projection of plane normal vector on yz plane of the CCF

7.2.5 Identification of all plane inliers and pixel labeling

As mentioned before, the criteria to crop the point cloud is very loose and it is possible that the actual plane may lie beyond the used horizon line if the camera is tilted towards the sea surface and therefore, the inliers found by the RANSAC are not complete. To overcome this issue, the distance (d_p) of every point in the raw point cloud is estimated from the estimated plane, and if this point lies within the threshold range ($dist\_th$) then it will be considered as an inlier.

$$d_p = \hat{n} \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} + d_\perp \tag{7.7}$$

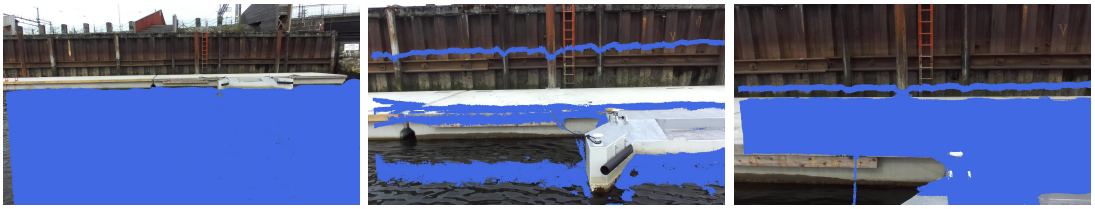
where x_p, y_p, and z_p represent the 3D point from the point cloud. As d_p is not always positive, it is possible to differentiate if the pixels are on the plane, above the plane, or below the plane.

$$label_p = \begin{cases} ABOVE\_PLANE, & \text{if } d_p < -d_{th} \\ ON\_PLANE, & \text{if } |d_p| \leq d_{th} \\ BELOW\_PLANE, & \text{if } d_p > d_{th} \end{cases} \tag{7.8}$$

Using eq. 7.8, I have labelled the 3D points according to their distances from the plane. I have used the same threshold value as RANSAC in-plane threshold (d_{th}) to determine if the point is an inlier or not.

7.2.6 Results and comments on RANSAC based plane segmentation

The above three steps were implemented and tested on the Z2-FM-EB-SEQ-1 sequence and the results are shown in Fig. 7.5. When the ferry was far from the dock (see Fig.. 7.5a), the water was dominating the lower half region of the image and therefore, the plane was successfully estimated but when the ferry approached the dock (see Fig.. 7.5b), the dock also became part of the input to the RANSAC plane estimator and resulted into a slanted plane which is inclined towards the dock. Finally, when the ferry was about to dock (see Fig.. 7.5c), the plane segmentor identified the dock as the water plane. From these results, it can be inferred that defining a constant horizon



(a) (b) (c)

Figure 7.5: Figures are taken from the sequence Z2-FM-EB-SEQ-1. The blue color on the graphics represents the inliers on the detected plane. This Fig. shows the problem of fitting a plane when there are objects below the horizon line. As we see, parts of the dock are considered as part of the water surface which gives wrong results. This problem has been addressed and solved in section 7.2.8.

line is not sufficient for robust plane estimation. Therefore, multiple checks and knowledge from the previous frame should be incorporated to estimate the ground/water plane.

7.2.7 How to prevent the method to lock into wrong planes

As we saw before, the method failed to find the water planes when it came closer to the dock and identified invalid or other planes as the water plane. If I pick the horizon line carefully or create a mask for all the water labels such that then the first plane can be safely assumed to be an actual water plane then there could be a number of potential solutions revolving around the first water plane to resolve the issue observed in Fig. 7.5b.

- **Adaptive cropping of point cloud**: The water plane information can be propagated to the next timestep to crop the raw point cloud such that the number of outliers is reduced.
- **Plane comparison:** The detected plane parameters (plane normal vector and the perpendicular distance to the camera) can be compared against the valid water plane. If the distance and the angle between the normal vectors of the detected plane and valid water plane should be within a range of uncertainty. Instead of measuring the angle between the normal vectors, the roll and pitch angle of the plane w.r.t the CCF can be computed as well and can be compared against the nominal roll and pitch angle (obtained from the valid water plane).

I have proposed some modifications in the RANSAC based plane segmentation approach and apply the proposed solutions to avoid the invalid water planes in the next section.

7.2.8 Adaptive Plane Segmentation using RANSAC (APSR)

From the section 7.2.6, it became clear that the point cloud data and the initial estimate of the horizon are not sufficient to identify the plane when the ferry is docking. To overcome this challenge, the point cloud should be cropped in an adaptive way, and the planes that don't identify as the ground plane should be eliminated. My proposed algorithm APSR consists of a bootstrapping step in which an approximate value of the horizon line is required at the beginning of the sequence to identify the water plane and then use this plane information in the subsequent frames to crop the point cloud. The plane is checked before identifying it as a water plane. Unlike the previous algorithm, the point cloud is cropped based on the adaptive inlier mask

that is updated after each iteration. This mask consists of the pixels that are the inliers of the detected water plane. The steps of the algorithm are the following.

- **Initialization:** The nominal plane is initialized as xz plane of the CCF and the initial inlier mask is identified using an initial guess of the horizon i.e. the xy plane of the CCF. The raw point cloud is cropped using this initial inlier mask followed by the fitting of the plane on the cropped point cloud and then a new inlier mask is created by labeling the pixels that lie on the fitted plane.

Algorithm 2 Plane initialization

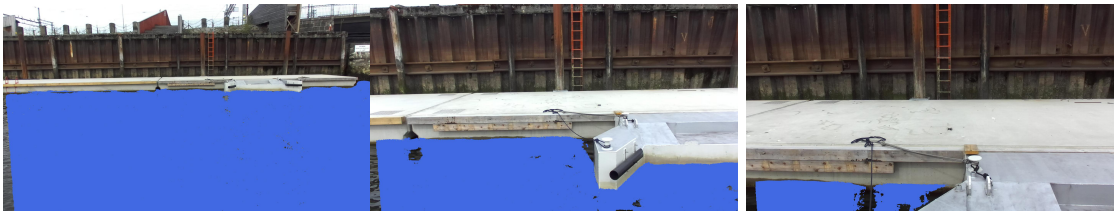
procedure $initialize\_plane(P^0)$
  $mask^0 = \begin{bmatrix} 0_{W \times H/2} \\ 1_{W \times H/2} \end{bmatrix}$
  $P'^0 = \textbf{crop}(P^0, mask^0)$
  $[\vec{n}^0, d_\perp^0] = \textbf{fit\_plane}(P'^0)$
  $mask^0 = \textbf{get\_label\_mask}(P^0, \vec{n}^0, d_\perp^0)$
  return $[\vec{n}^0, d_\perp^0, mask^0]$

- **For every next frame:**
  1. *Cropping and plane fitting:* The previous inliers mask is used to crop the point cloud followed by the plane identification.
  2. *Compliance check 1:* The fitted plane is validated against the previous estimated plane. If the normal vector from the current and previously estimated plane is making an angle less than *angle_th* and the difference between the height of the camera from the current and previous plane is less than *height_th* meters, then the plane is marked as valid and invalid otherwise (see table 7.3 for the threshold parameters).

Algorithm 3 Plane compliance check

procedure $is\_valid\_plane(\vec{n}^t, d_\perp^t, \vec{n}^{t-1}, d_\perp^{t-1})$
  $height = |d_\perp^t - d_\perp^{t-1}|$
  $angle = \arccos(\vec{n}^t \cdot \vec{n}^{t-1})$
  if $angle \leq angle\_th$ *and* $height \leq height\_th$ **then**
    return $True$
  else
    return $False$

  3. *Compliance check 2:* The fitted plane is validated against the nominal plane (the plane identified in the beginning).
  4. *Labelling using nominal plane:* If the fitted plane failed the compliance check 1 or compliance check 2, then the inlier mask is created using the nominal mask instead.



(a) (b) (c)

Figure 7.6: Figures are taken from the sequence Z2-FM-EB-SEQ-1. The blue color on the graphics represents the inliers on the detected plane. This Fig. shows the improved results of the improved water surface detection (compared to Fig. 7.5), where the prior knowledge is also used

The steps of the algorithm are summarised in alg. 4. From Fig. 7.6, it can be seen that the adaptive masking of the inliers and the consistency checks enabled the algorithm to find the water plane even during the docking and the false water plane observed in Fig. 7.5b and Fig. 7.5c are fixed using APSR. The major drawback of this proposed algorithm is that it can not recover from a non-water plane zone. Once the ferry completely docks, the inlier mask will be empty and the algorithm can not recover the plane when the ferry undocks.

Algorithm 4 Adaptive plane segmentation using RANSAC

```
1  procedure adaptive_plane_seg()
2      [n⃗^0, d_⊥^0, mask^0] = initialize_plane(P^0)
3      for t = 1 to ∞ do
4          P'^t = crop(P^t, mask^t)
5          [n⃗^t, d_⊥^t] = fit_plane(P'^t)
6          if is_valid_plane(n⃗^t, d_⊥^t, n⃗^{t-1}, d_⊥^{t-1}) then
7              if is_valid_plane(n⃗^t, d_⊥^t, n⃗^o, d_⊥^o) then
8                  mask^t = get_label_mask(P^t, n⃗^t, d_⊥^t)
9              else
10                 mask^t = get_label_mask(P^t, n⃗^o, d_⊥^o)
11         else
12             mask^t = get_label_mask(P^t, n⃗^o, d_⊥^o)
```

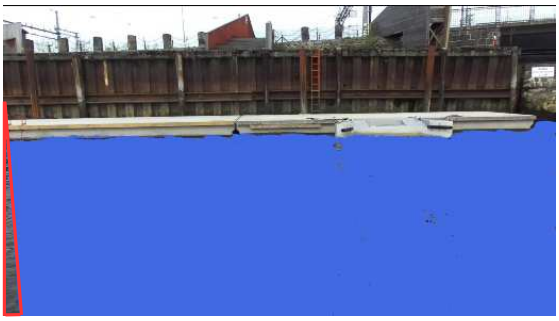


Figure 7.7: Disparity free zone from RANSAC based plane segmentation

7.3 Limitations of the proposed plane segmentation algorithms

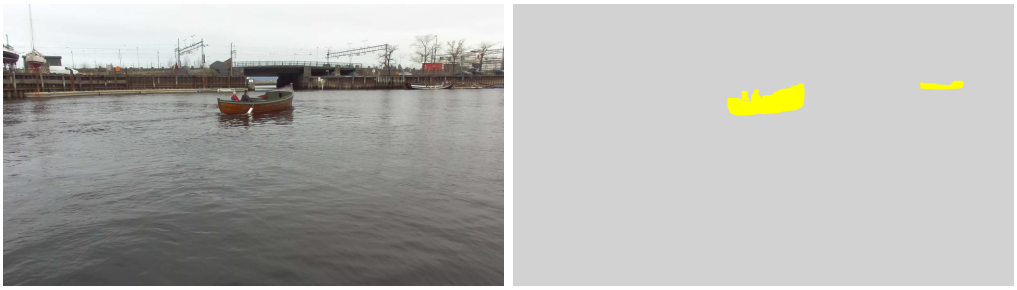
Both approaches have their pros and cons with respect to the application. In the Aquanet-based approach, the network can successfully identify the water region using only the image. In most cases, it is able to identify the whole region of water without any holes which is not possible in RANSAC-based approach because of its strong dependency on the disparity data. From Fig. 7.7, it can be seen that the APSR algorithm failed to label the pixels in the disparity-free area. These disparity-free areas will always exist because the overlapping between the left and right images of the stereo pair is not 100% complete. The holes in the water plane found by neural network-based approaches may suffer from the reflections of the land structures and in the geometry-based method, inaccuracy in the disparity estimates can lead to the holes in the segmented water plane.

7.4 Object detection using YOLO-v8

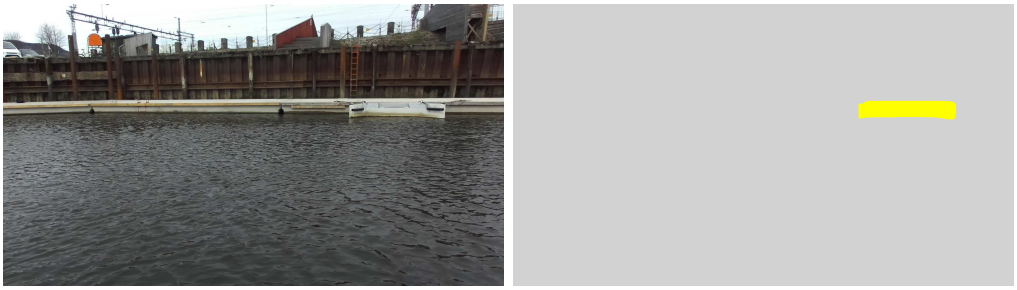
[37] introduced YOLO (You Only Look Once) which is one of the most popular Convolutional NN for object detection in real-time in 2015. Over the years, multiple versions of the YOLO have been released and a complete review on them is given by [45]. I decided to use the latest version of the YOLO (YOLO-v8[3]) to segment the boats and other objects from the scene. The YOLO-v8 comes in different sizes which means that the size of the model differs according to application. For limited GPU memory, the nano size (*yolov8n-seg*) of the YOLO-v8 can be used and for excessive GPU memory, the extra large model (*yolov8x-seg*) can be used. The accuracy and performance increase with the size of the network therefore, the extra-large model has the best performance among all of the trained models. In this project, Taking the available GPU memory into consideration, I decided to use the largest version *yolov8l-seg* for the segmentation.

Figure 7.8 shows the results of the YOLO-v8 network on the recorded sequences. In Fig. 7.8b, we can see that the network found both of the boats in the scene even the farthest one but at the same time, the network also suffers from the false positives as shown in Fig. 7.8d where YOLO identified the dock as the boat.

[3]The link for the implementation of YOLO-v8 is here



(a) Image from seq. Z2-FM-EB-SEQ-2. (b) YOLO mask for boats only.



(c) Image from seq. Z2-FM-EB-SEQ-1. (d) YOLO mask for boats only.

Figure 7.8: The input and the output to and from the YOLO-v8 network. The output label image is colored according to the color coding defined in table 7.2.

Table 7.4: IMOs' class labels and their values in YOLO-v8

class	class value in YOLO-v8
person	0
bicycle	1
car	2
motorcycle	3
bus	5
train	6
truck	7
boat	8

7.5 Computation of a Semantic Image by Fusion of Different Algorithms

In this chapter, we saw different strategies that can be used to segment the water plane and the boats from the image. Each proposed segmentation algorithm whether it is Aquanet or APSR, or the YOLO-v8 has its own pros and cons. The fusion of all of these proposed semantic segmentation algorithms can provide more reliable and complete knowledge of the water surface and the boats, therefore, in this section, I will propose the methodology to fuse the information provided by them.

The algorithm to fuse the masks from different algorithms is given in algorithm 5. Initially,

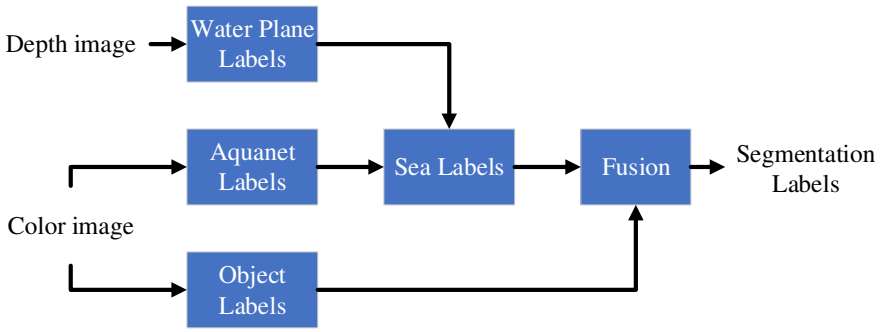
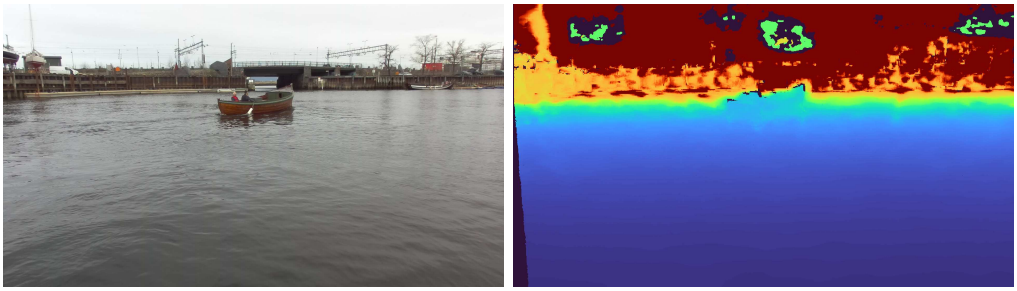


Figure 7.9: Semantic labeling of the environment

the YOLO-v8 semantic segmentation network also referred to as the *object_detector* is used to detect the object/boat in the sequence. Also, the APSR based *plane_segmentor* is used to find the pixels in the image that lie on the ground plane or on the water plane in the maritime sequence and finally, I am getting a semantic label mask, similar to the one provided by the YOLO-v8, that contains the information of the pixels that are on the sea from the Aquanet and referred as *sea_segmentor*. The *plane_mask* from the *plane_segmentor* is merged with the *aquanet_mask* from the *sea_segmentor* such that the pixels have been marked with *sea* label in both masks will be remarked to the label *definite_sea* and the pixels which are not marked in both but only in once are remarked to the label *maybe_sea*. The *sea_mask* is overlayed with the *object_mask* such that the pixels in the *fused_mask* will be marked *boat* even if they are marked with *definite_sea* in the *merged_sea_mask* to prioritize the boat identification over the sea. The inputs, intermediate, and final results of the algorithm are shown in Fig. 7.10.

Algorithm 5 Fusion of different semantic and plane segmentation masks

```
#Define EPS = 0.000001
procedure fuse_masks(depth_img, color_img, intrinsics)
    object_mask = object_detector(color_img)
    plane_mask = plane_segmentor(depth_img, intrinsics)
    aquanet_mask = sea_segmentor(color_img)
    sea_mask = merge_masks(plane_mask, aquanet_mask)
    fused_mask = overlay(sea_mask, object_mask)
      return fused_mask
```



(a) Input color image.

(b) Input depth image in false colors.

(c) Aquanet mask

(d) Plane mask.

(e) Object mask

(f) Sea mask.

(g) Fused mask

Figure 7.10: Input and output of the algorithm 5. Image taken from Z2-FM-EB-SEQ-2 sequence is used as an input and the output label image is colored according to the color coding defined in table 7.2.

CHAPTER 8

Labeling Pixels as belonging to Upright or Horizontal Structures

Contents

In the previous chapter, we saw how the Adaptive Plane Segmentation using RANSAC (APSR) method can find the water plane. If the same algorithm is run recursively, it is possible to estimate multiple planes from the point cloud. I tried to do the same (see appendix B, p.175) and found out that each plane has different characteristics and fits the plane using the RANSAC with the same set of parameters as of the water plane is not ideal. Also, if there is no dominant plane (i.e. the number of inliers is smaller than the number of outliers) at any iteration while finding the planes recursively then the estimated planes can be wrong.

In this chapter, I will explain a more simple but effective approach to classify pixels whether they lie on a horizontal or an upright surface in a single iteration. The significance of the upright pixels is that any region behind the upright pixels is an obstacle and the ego-vehicle can not reach it if it keeps moving straight towards it.

8.1 The principle of detecting upright or horizontal structures from the disparity image: the 'Stixel Principle'

The *stixels* is an efficient way to represent the environment ([1]). The representation assumes that the objects in the surroundings can be approximated using only the vertical and horizontal

surfaces. Instead of representing the objects using the thousands of 3D points from the point cloud, they can be represented using a set of adjacent rectangular strips called stixels (see Fig. 8.1a).

In this project, I also wanted to use the stixels to represent the environment, therefore, I implemented the most basic simplified version which is based on the stixel principle. I extracted the horizontal and upright surfaces using the disparity data only. If I assume that the camera is looking parallel to the ground plane and every object in the scene is a box lying on the ground plane, then their disparity will be constant for the upright surfaces of the object (if the surface is perpendicular to the viewing direction) whereas the disparity will be increasing (when we go from top to the bottom of the vertical disparity profile) for all horizontal planes (the farthest point on the plane will have less disparity as compared to the closest point). In other words, for a column in the disparity image, the vertical line segments (constant disparity) represent the upright surfaces, and the slanted line segments (constantly changing disparity) represent the horizontal surfaces (see Fig. 8.1b).

8.2 Looking at vertical profiles of disparity

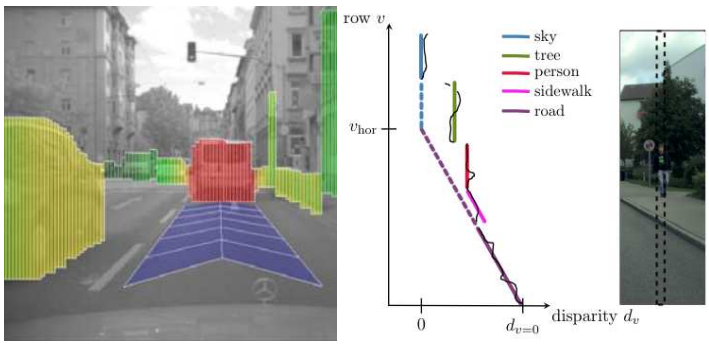
In the previous section, I discussed the shape of the vertical disparity profile for upright and horizontal surfaces but in this section, I will try to prove the same mathematically followed by some real examples.

Let c be the distance of the plane under consideration, then the equation of the plane w.r.t the Camera Coordinate Frame (CCF) ($y - axis$ is pointing downwards and $z - axis$ is pointing forwards) parallel to the camera's viewing direction ($z - axis$) is given by

$$y = c \tag{8.1}$$

Let $\vec{p}$ be a 3D point with coordinates $[x, y, z]^T$ w.r.t the CCF and $\vec{p}_h$ be the projection of point $\vec{p}$ on the plane $y = c$, then the point $\vec{p}_h$ lying on the plane $y = c$ can be easily known.

$$\vec{p}_h = \begin{bmatrix} x \\ c \\ z \end{bmatrix} \tag{8.2}$$



(a) (b)

Figure 8.1: The left picture shows the representation of the environment using stixels (Taken from [1]) and the right picture shows the variation of the disparity along the vertical column of the image (Taken from [10]).

Let d_h be the Euclidean distance of the point $\vec{p}_h$ from the CCF.

$$\begin{aligned} d_h &= \sqrt{\vec{p}_h^T \cdot \vec{p}_h} \\ &= \sqrt{x^2 + c^2 + z^2} \end{aligned} \tag{8.3}$$

Let α_h be the slope between the distance d_h and the depth z of the point p_h, then the slope α_h can be computed by taking the partial derivate of distance d_h w.r.t depth z.

$$\begin{aligned} \alpha_h &= \frac{\partial d_h}{\partial z} \\ &= 1 \end{aligned} \tag{8.4}$$

From the above equation, it can be stated that if there is a point $\vec{p}_h$ on the plane $y = c$ such that it is going away from the CCF (z is increasing), then the distance d_h also increases with z because of the constant slope $\alpha_h = 1$ or the disparity decreasew with the depth z because it is inversely proportional to the depth z. In other words, The disparity has a negative slope with respect to the distance of point $\vec{p}_h$ lying on a horizontal plane which is also observed in Fig. 8.1b.

Let us assume another plane that is perpendicular to the viewing direction of the camera, then it can be expressed using the following equation.

$$z = c \tag{8.5}$$

Let $\vec{p}_v$ be the projection of point $\vec{p}$ on the plane $z = c$.

$$\vec{p}_v = \begin{bmatrix} x \\ y \\ c \end{bmatrix} \tag{8.6}$$

Let d_v be the Euclidean distance

$$d_v = \sqrt{x^2 + y^2 + c^2} \tag{8.7}$$

Let α_v be the slope between the distance d_v and the depth z of the point $\vec{p}_v$,

$$\begin{aligned} \alpha_v &= \frac{\partial d_v}{\partial z} \\ &= 0 \end{aligned} \tag{8.8}$$

It is clear that the distance d_v is independent of the depth z when the point $\vec{p}_v$ is lying on the upright plane. It also implies that the disparity remains constant irrespective of the position of point $\vec{p}_v$ on the plane $z = c$ and hence, it is the reason why a vertical line (constant disparity) can be observed for the upright surfaces (see Fig. 8.1b).

Figure 8.2 shows different vertical profiles of disparity data[1]. In figure 8.2b, we can see that the slanted line segment after the 450^{th} value in the $y - axis$ of the plot because of the horizontal water plane and a dominating vertical line segment due to the wall on the back. A small vertical line segment around the 450^{th} value in the $y - axis$ due to the front wall of the platform. The same small vertical line segment is shifted right for the dock (see red line in the plot) because the dock is extruded from the platform and therefore, it is more closer to the camera (the more will be its disparity) and hence it is shifted right.

[1]It should be noted that I used the Neural method of the ZED camera to compute the disparity/depth data in this chapter instead of the Ultra mode because the depth images from the Neural mode are more complete and can provide a better surface estimation.

Figure 8.2e shows the vertical disparity profiles when the ego-vehicle is closer to the dock. The vertical line segments for the front wall of the platform and the wall on the back are slightly tilted because of the pitch of the camera. The effect of the pitch was not dominant or clearly visible when the wall was far away. For the red line, we can three slanted line segments. From the bottom, the first slanted line (coinciding with blue and green line segments) represents the water plane. The second slanted line segment between the 600^{th} and 800^{th} value in the $y-axis$ represents the slanted plane of the dock and finally, the third slanted segment shows the combined horizontal surface of the platform and the dock. It can be observed that the slope of the slanted line segment for the dock is different than the water plane and the platform. It implies that it is possible to extract only the horizontal and vertical surfaces using the vertical disparity profiles.

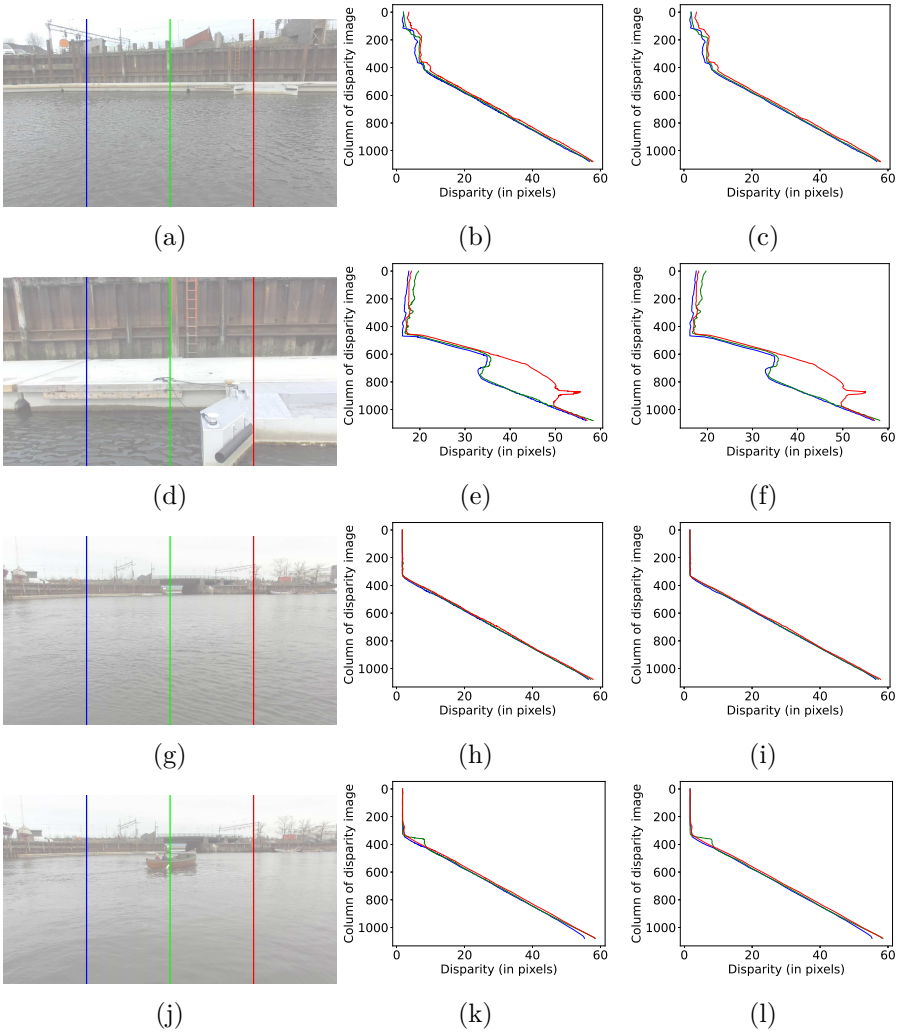
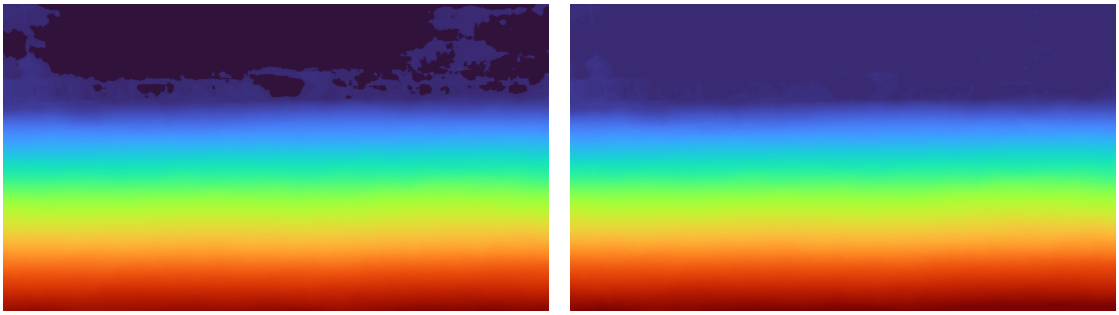


Figure 8.2: The left images are from the recorded sequences and highlight the vertical columns under consideration. The middle and right figures show the vertical disparity profiles before and after the low-pass filtering respectively.



(a) (b)

Figure 8.3: The left and right images the disparity profile (in false colors) before and after the linear interpolation respectively.

8.3 Differentiation between the "upright" and "horizontal" surfaces

The slope of the line segments in the vertical disparity profiles can help us to identify different types of surfaces. Therefore, I computed the derivative on these vertical disparity profiles. The disparity image from the ZED camera contains NAN, INF, and $-INF$ for different cases in which the disparity couldn't be computed (see section D.2, p.184), therefore, I performed 1D interpolation on the raw disparity data to fill the invalid values with real numbers before proceeding to the derivatives.

Let d be the disparity image of width W and height H, i be the vertical column in the disparity image, $d(i)$ be the disparity data in the column i, idx_i be an array containing the row index in the column i which has the valid disparity data, then we can extract the row indices from the column i.

$$idx_i = \{j \text{ if } d(j,i) \text{ is valid for } j = 0...H\} \tag{8.9}$$

Let $valid\_d(i)$ be the disparity value for the idx_i such that the disparity value stored in $valid\_d(i)$ are always valid. It should be noted that the values in idx_i are monotonically increasing, therefore, I can use the *interp* function from the Numpy library to perform the 1D interpolation on the disparity data $d(i)$. Let $d_I(i)$ be the interpolated disparity data for i^{th} column, then I can use the *interp* function on $valid\_d(i)$ for the linear interpolation.

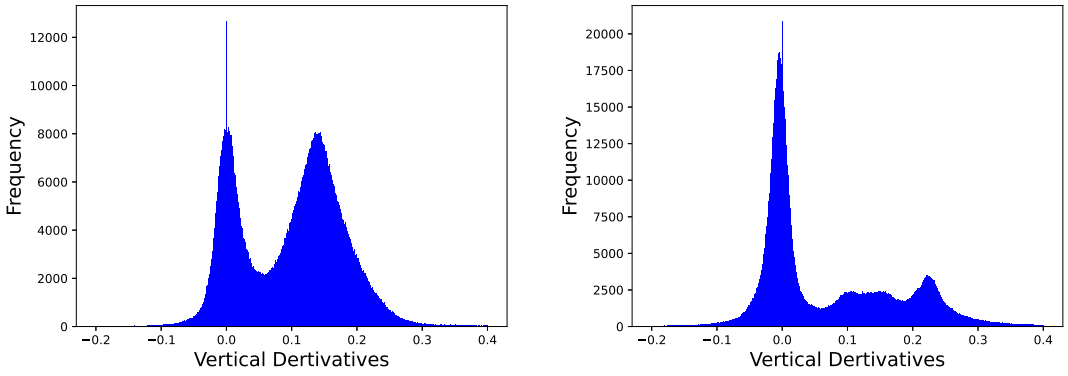
$$d_I(i) = interp(idx_i, valid\_d(i)) \tag{8.10}$$

The derivatives are very sensitive to noise, therefore, I first applied the low-pass filter on the interpolated disparity profile d_I. Let K be a kernel of size $n \times 1$, and d_f be the low-pass filtered disparity image, then the low-pass filtering can be performed by convolving the kernel K with i^{th} column of the disparity image d_I.

$$d_f(i) = K * d_I(i) \quad \forall\ i = 0...W \tag{8.11}$$

where kernel K is defined as

$$K = \begin{bmatrix} 1/n \\ ... \\ 1/n \end{bmatrix}_{n \times 1} \tag{8.12}$$



(a) (b)

Figure 8.4: The left and right figures show the histograms of the vertical derivatives of the disparity image when the ego-vehicle was far from the dock (see Fig. 8.2a) and close to the dock (see Fig. 8.2d) respectively. The lower and the upper limits of the histogram have been clipped.

Let M be the 1D Sobel kernel of size 3×1, and ∂d be the vertical derivative image of the low-pass filtered disparity image d_f, then the vertical derivative image can be computed by convolving the kernel M with each column of the low-pass filtered disparity image d_f.

$$\partial d(i) = M * d_f(i) \quad \forall\ i = 0...W \tag{8.13}$$

where kernel M is defined as

$$M = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \tag{8.14}$$

As mentioned before, each value of the slope corresponds to a particular type of surface (horizontal, upright, slanted, etc.), the histogram of the vertical derivatives can show the number of different surface orientations. Figure 8.4a shows the histogram of the derivative values in the vertical derivative image ∂d when the ego-vehicle was far from the dock (see Fig. 8.2a). There are only two dominating surfaces (horizontal and upright) in this case, therefore, two peaks can be observed in the histogram. The peak near the zero value of the vertical derivative corresponds to the upright surfaces and the other peak corresponds to the horizontal surface (water plane), however, when the ego-vehicle was close to the dock (see Fig. 8.2d), the upright surface (wall in the background) became more dominant (the peak at zero derivative value got bigger in Fig. 8.4b) but the other peak in the histogram is not clearly visible as in the previous case because of the presence of other slanted surfaces in the scene such as the slanted surface of the dock.

The base for the classification of the pixels of the image is to look at the derivatives of their disparity. If the derivative for a particular pixel lies around a peak as shown in the histogram, then it can be labeled accordingly. I classified the pixels into three categories: (1) *UPRIGHT*, (2) *HORIZONTAl*, and (3) *UNKNOWN*. The labeling of the pixels if the derivative of their disparity is around zero as UPRIGHT is not enough as a hard threshold on the limits of the derivatives can lead to incorrect results, especially for the edge cases. To find the range thresholds for the derivatives, I represented each peak using a Gaussian curve. Let μ be the mean, and σ be the

Table 8.1: Parameters used in the identification of Upright and Horizontal surfaces.

Parameter	value
Low-pass kernel filter size (n)	5
Lower clipping limit for clustering	-0.3
The upper clipping limit for clustering	0.5

standard deviations of the fitted Gaussian curves such that the lower ∂d_L and upper ∂d_U range bounds for the derivatives can be calculated from Gaussian parameters.

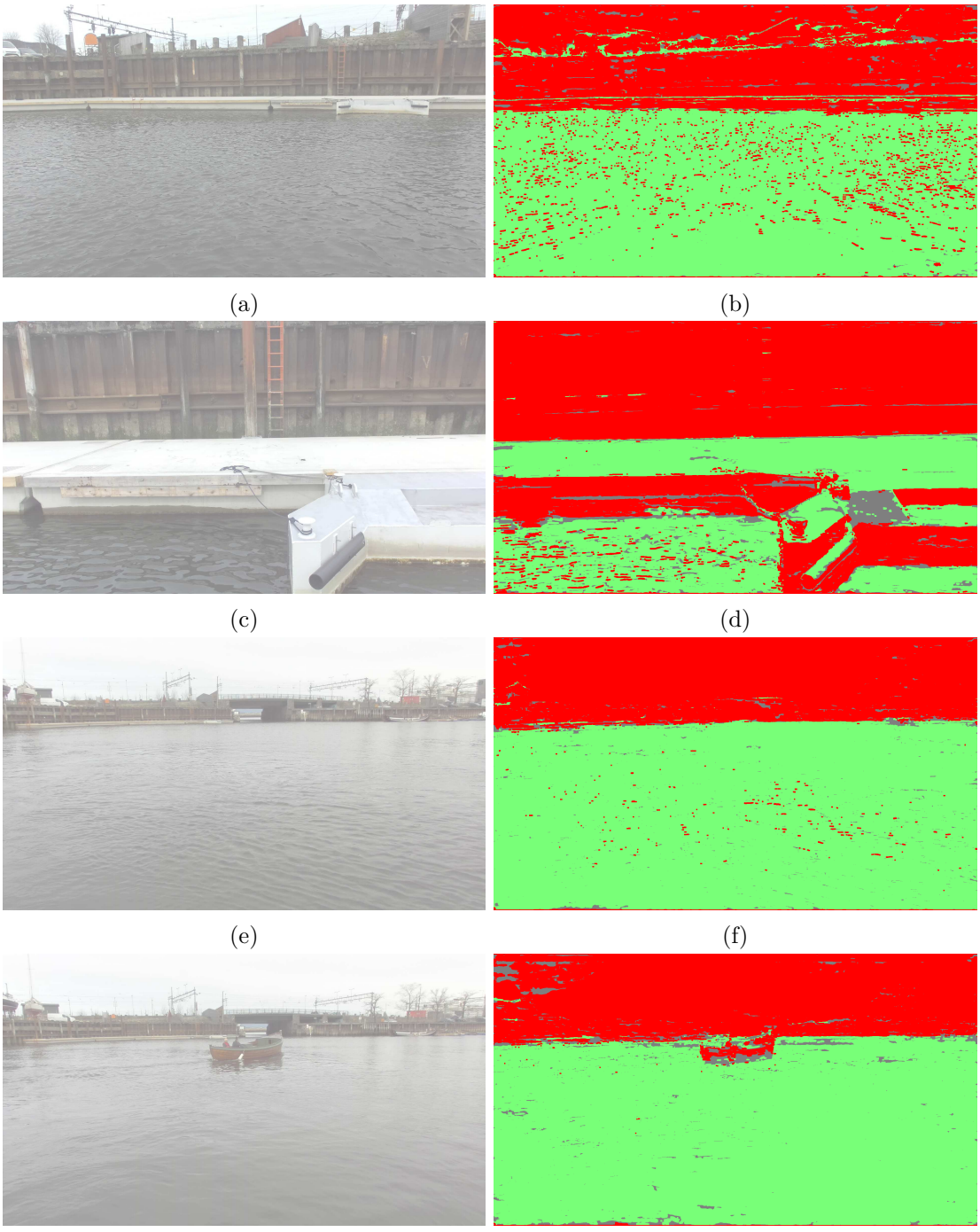
$$\begin{aligned} \partial d_L &= \mu - \sigma \\ \partial d_U &= \mu + \sigma \end{aligned} \tag{8.15}$$

As we saw in Fig. 8.4, I may have one or more than one peak in the histogram data but the classes I am mostly interested in are the *UPRIGHT* and *HORIZONTAL*. Therefore, I fitted two Gaussian curves to the derivative data. To fit the curve, I need to know which derivative value belongs to which Gaussian curve, and to know the same, I fit two clusters in the derivatives data using *K-means*[2] such that each cluster's center may lie on the peak. I observed that the derivative data has extremely low and extremely high values which influenced the clustering algorithm negatively. Therefore, I decided to clip the lower and upper ranges for the derivatives data. The algorithm also requires the initial estimate of the center of the cluster. From the histogram data, I approximately pick the derivative values where the peak lies and used them for the initial estimate of the peak locations. After fitting the derivative data into two clusters, I fitted the Gaussian to each cluster. Let μ_1 and μ_2 be the means and σ_1 and σ_2 be the standard deviations of the fitted Gaussian curves, $\partial d_L 1$, $\partial d_L 2$ be the lower and $\partial d_U 1$, $\partial d_U 2$ be the upper range limits for the first and second curve respectively, then the range limits can be computed for each curve using the eq. 8.15. It is possible that the upper range limit of the left peak overlaps with the lower limit of the right peak. To handle this case, the priority of the label should be defined. In my case, the class UPRIGHT will always be the left peak with its mean near zero (the upright surfaces have 0 slope/derivative) and they should be given higher priority as they are potential obstacles that should be avoided at all cost. Let $L(x, y)$ be the label that defines the class of the pixel (x, y), then it can be created using the derivative image and the fitted Gaussian curves.

$$L(x,y) = \begin{cases} HORIZONTAL, \text{if } \partial d_{L2} < \partial d(x,y) < \partial d_{U2} \\ UPRIGHT, \text{if } \partial d_{L1} < \partial d(x,y) < \partial d_{U1} \\ UNKNOWN, \text{otherwise} \end{cases} \tag{8.16}$$

In the above expression, if the pixel is labeled as *HORIZONTAL* then the condition for the *UPRIGHT* should be checked as well to give priority to this class and not the vice-versa.

[2] I used sklearn library for the implementation of K-means algorithm. The link to the function is here.



(a) (b) (c) (d) (e) (f) (g) (h)

Figure 8.5: The left and the right figure show the input and the upright-horizontal labeled images respectively. The pixels colored green are labeled as *HORIZONTAL*, red as *UPRIGHT*, and gray as *UNKNOWN*.

Chapter 9

Computation of a Depth Birds Eye View

Contents

The depth map obtained from the stereo-matching algorithm can never be trusted completely as the accuracy of the depth estimate vanishes when the point goes to infinity. The depth accuracy highly depends on the stereo-matching accuracy (assuming the intrinsic and extrinsic calibration parameters of the stereo camera are absolutely correct which is not true either). The representation of the depth image in false colors doesn't tell us much about the noise in it and therefore, the most intuitive way is to represent the point cloud (generated from the depth map) in the Bird's Eye View (BEV). When we look at the point cloud data from the BEV, we can see how the depth is distributed along the boundaries of the objects in the scene. In an ideal case, if there is a wall in the camera's image then it should be represented as a straight line in the BEV (see Fig. 9.1), however, due to the noise in the depth data, the line will be deformed.

In this chapter, I will talk about the fundamental principles of the BEV followed by a depth filtering technique to reduce the noise in the depth maps and finally the results of the BEV.

9.1 Bird's Eye View

In the BEV, I used the orthogonal view (rather than the perspective view) that is perpendicular to the camera's view and it is looking towards the ground plane, i.e. if $[x, y, z]$ are the coordinates of a 3D point in the Camera Coordinate Frame (CCF), I reduced them to the $[x, z]$ coordinates. In the advanced modes, I first transformed the 3D coordinates of the pixels from the CCF to some world coordinate frame for consistency in the BEV. How the points with the same $[x, z]$ values

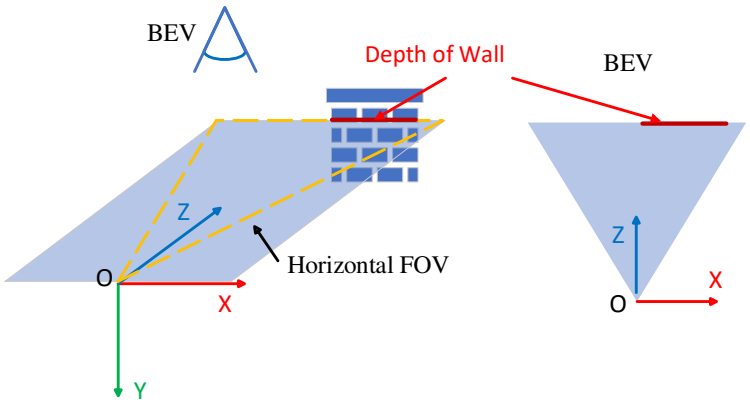


Figure 9.1: Explanation of the BEV.

but different y values are dealt with is explained in section 9.1.3. In the following sections, I will discuss the steps to prepare a BEV that can be used to analyze the depth and the semantically labeled data together.

9.1.1 Generation of a point cloud

Let $[x_{px}, y_{px}]$ be the pixel coordinates of any 3D point $[X, Y, Z]$ in the space and D be the depth of this point in the CCF, then the 3D coordinates of the point can be computed from the pixel coordinates, depth, and the intrinsic parameters of the camera.

$$\begin{aligned} X &= \frac{(x_{px} - c_x)D}{f_x} \\ Y &= \frac{(y_{px} - c_y)D}{f_y} \\ Z &= D \end{aligned} \tag{9.1}$$

9.1.2 Cropping the point cloud

The point cloud generated from the stereo data is w.r.t CCF and if we want to find the navigable area or the corridor that the vehicle can pass through, we need to transform the point cloud from CCF to the ground (water) plane coordinate frame by first compensating the roll and pitch of the camera w.r.t the water plane such that the $x - z$ plane of the CCF is aligned with the water plane and then we need to compensate the height of the vehicle such that we ignore all the 3D points that are lying above the vehicle. In this project, I compensated for the pitch angle and assumed that the roll angle is negligible, and picked a random value for the height of the ego-vehicle to compensate for the height. In an actual implementation, the roll, pitch, and height parameters can be obtained from the external calibration and then compensated before computing the BEV.

When the ego-vehicle is moving on the water surface with the camera at a given height, the points below the water surface may not appear or are irrelevant, and the points that are much higher than the vehicle's own height (e.g. bridges) are not relevant either. Also, in an outdoor scene, the point clouds can have points at greater depth and such points can be ignored

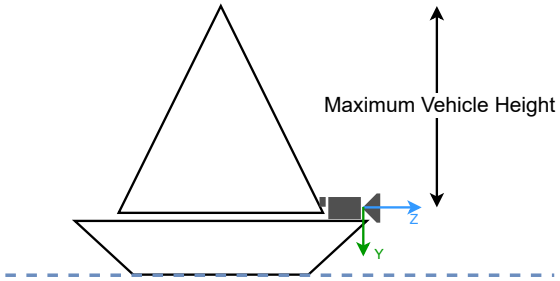


Figure 9.2: Maximum height threshold for the ego-vehicle

in the BEV. Taking these points into consideration, I decided to crop the point cloud prior to generating the BEV. It should be noted that the cropping of point cloud data has been done solely for the generation of the BEV's images and has no influence on the ego-motion estimation sub-system.

Instead of using the range thresholds for each Cartesian axis in the CCF, I used only the maximum depth and maximum height threshold. The vertical threshold makes sure that the points are not too high from the camera. The other thresholds are computed depending on the properties of the camera.

Let D_{th} be the maximum depth threshold, fov_h be the horizontal Field of View (FOV), min_x be the minimum range threshold for the $x-axis$, and max_x be the maximum range threshold for the $x-axis$, then the minimum min_x and maximum max_x range thresholds can be computed using the horizontal FOV fov_h.

$$\begin{aligned} min_x &= -D_{th} \tan\left(\frac{fov_h}{2}\right) \\ max_x &= D_{th} \tan\left(\frac{fov_h}{2}\right) \end{aligned} \tag{9.2}$$

There is no threshold on the upper limit for the y component of the point and for the $z-axis$ the upper range threshold is given by D_{th} and no bound on the lower range as it is inherently bounded by the constraint of having non-negative values. The only parameter that is left unchecked is the lower range for the minimum value of y (maximum height threshold).

Pitch compensation in the BEV

It should be noted that the positive $y-axis$ of the CCF is pointing downwards, therefore, I am mostly interested in putting a threshold on the minimum value (min_y) that a point can have in the $y-axis$ such that the maximum height can be maintained in the BEV. The maximum height of the points in the BEV can be maintained only if the camera is looking parallel to the ground plane but if there is some pitch in the camera due to the waves, then it has to be either compensated in the maximum height threshold or in the whole point cloud. The limitation of the former approach is that the depth threshold D_{th} has to be adapted according to the scene. For a close-range scene, if I use a large depth threshold D_{th}, then for a given pitch angle, the maximum height threshold will be over-estimated. Therefore, I used the latter approach to avoid this issue and compensate for the pitch in the point cloud directly.

Table 9.1: Constant parameters for the cropping of point cloud

Parameter	code name	Value
Depth threshold (D_{th})	D_th	50 meters
Maximum height threshold (y_{th})	y_th	-2 meters
BEV's image width (W)	bev_w	800 pixels
BEV's image height (H)	bev_h	1000 pixels

Let θ be the pitch angle of the camera w.r.t. the plane and $\mathbf{R}_x(\theta)$ be the 3D rotation matrix for a rotation of θ angle about the $x - axis$.

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{9.3}$$

Let P be the input point cloud and P' be the pitch compensated point cloud, then the pitch compensated point cloud can be computed by pre-multiplying the rotation matrix $\mathbf{R}_x(\theta)$ with each point from the input point cloud P.

$$P' = \{p' = \mathbf{R}_x(\theta) \times p \quad \forall p \in P\} \tag{9.4}$$

The pitch angle θ can be computed for each new frame with the help of Adaptive Plane Segmentation using RANSAC (APSR) module (see section 7.2.4, p.45).

9.1.3 Generation of the BEV

Once, the point cloud is cropped and the pitch is compensated, I mapped the points from the Cartesian coordinates to the pixel coordinates. Let W and H be the width and height of the BEV's image, x_{px} and y_{px} be the x^{th} and y^{th} pixel coordinates in the BEV's image, then the x and z components of the 3D point can be mapped to the x_{px} and y_{px} pixel coordinates by first scaling them and then moving the origin of the CCF to the bottom center of the image.

$$\begin{aligned} x_{px} &= \frac{W}{2}\left(1 + \frac{x}{max_x}\right) \\ y_{px} &= H\left(1 - \frac{z}{D_{th}}\right) - 1 \end{aligned} \tag{9.5}$$

All of this is implemented using *Numpy* library in Python so the computational time to compute each BEV's image is insignificant. It is possible that multiple points from the point cloud may fall on the same pixel in the image. For example, the outer curvature of the boat and the water just below it will have the same x and z components but their y component will be different. In such scenarios, the point which is highest above the ground i.e. with the lowest y value ($y - axis$ is pointing downwards in the CCF) shall take the position in the image. This will make sure that the points that are not visible from the BEV are suppressed. I didn't use this approach because the computational time to compute each BEV was in the order of seconds due to the $W \times H$ number of iterations.

9.1.4 Colouring of the BEV's image

The final step to generate the BEV's image is to color each pixel according to its semantic label that we obtained from the fusion of different semantic segmentation masks (see section 7.5, p.51).

I used the color coding defined in table 7.2 to represent the segmentation results in the BEV's image as well to avoid confusion.

9.2 Temporal Filtering of the disparity data

The accuracy in the depth data is tightly coupled with the accuracy of the stereo matching algorithm which further depends on the pixel noise ([16]) and therefore, the depth value for a pixel can be different for every new stereo pair even if the camera and the object are not moving. The most effective and simple way to suppress such noise is to use the temporal filter. In the simplest temporal filtering technique, we take the average of the depth images over a time window, however, the depth images are computed using the disparity data, therefore, I implemented the temporal filter on the disparity data.

Let $d(x, y, k)$ be the disparity of the pixel (x, y) in an image taken at timestamp k. It should be noted that $d(x, y, k)$ may contain invalid disparity data due to occlusions, etc., therefore, the invalid disparity data should be filtered before proceeding to the temporal filtering. All these invalid values have been replaced with some non-zero EPS instead of zero because if it is replaced with zero, then the depth can not be computed for that pixel.

$$d(x, y, k) = \begin{cases} d(x, y, k), & \text{if } d(x, y, k) \textit{ is valid} \\ EPS, & \text{otherwise} \end{cases} \tag{9.6}$$

Let $d_f(x, y, k)$ be the temporal filtered disparity for the pixel (x, y) and L be the temporal window length then the temporal filtered disparity $d_f(x, y, k)$ can be computed as follows.

$$d_f(x, y, k) = \frac{1}{L} \sum_{i=k-(L-1)}^{k} d(x, y, i) \quad \forall k \geq L \tag{9.7}$$

The disparity data also contains the spatial noise apart from the temporal noise, therefore, I used the low pass filter of size $K \times 1$ in the vertical direction of the temporal filtered disparity image $d_f(k)$. Let f_x be the horizontal focal length of the left camera and b be the baseline of the stereo setup, and $D_f(x, y, f)$ be the temporal filtered depth for pixel coordinate (x, y), then $D_f(x, y, f)$ can be computed from the temporal filtered disparity $d_f(x, y, k)$.

$$D_f(x, y, k) = \frac{f_x b}{d_f(x, y, k)} \quad \forall k \geq L \tag{9.8}$$

9.3 BEV of temporally filtered depth data

To color the pixels in the BEV, I need to label each pixel in the image and then color the pixels according to the label. In chapter 7 and chapter 8, we saw two different strategies to label the pixels. One was based on the semantic labeling and the other was based on the classification of the labels based on their location on the upright and horizontal surfaces.

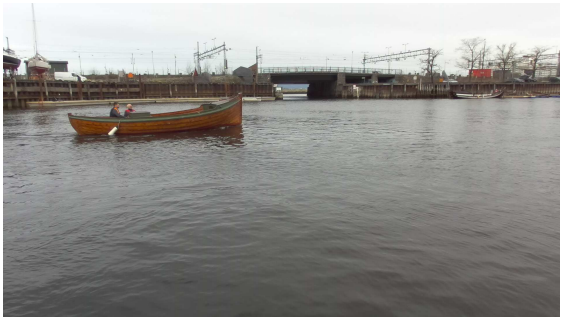
Let d be the disparity image after filtering the non-valid values (see eq. 9.6) and d_buffer be the memory buffer containing previous $L-1$ disparity images, then the depth's BEV can be computed using algorithm 6 in which the function *get_pixel_labels* returns the label image from either the semantic labeling or the upright-horizontal surface labeling.

Figure 9.3d and Fig. 9.3e show the depth's BEV for some frame in sequence Z2-FM-EB-SEQ-2 (see Fig. 9.3a) using the semantic labeling and the upright-horizontal surface labeling

respectively. In the fused mask (see Fig. 9.3b), we can see two boats in yellow color but in the BEV, we only say one boat in yellow color because the other boat is farther than allowed depth threshold D_{th}. Also, the density of the sea labels in the BEV reduces when we go far from the camera. Finally, the region behind the boat is not visible in the image and therefore, a gray void can be observed behind the boat in the BEV. The similar observations can be made for the BEV generated using the upright-horizontal surface labels. The boat is colored red because it is an approximately upright surface whereas the sea is colored green because it is on the horizontal surface. In both the BEVs, thin horizontal lines can be observed that split the total length of the BEV into equal partitions to visualize the approximate depth of each observed object.

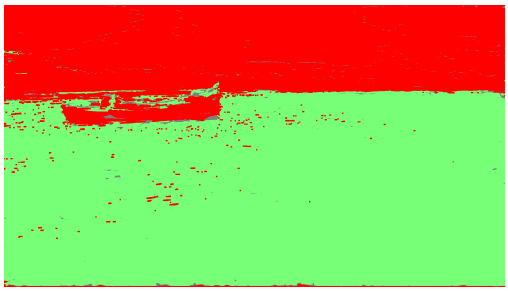
Algorithm 6 Preparation of Depth Birds Eye View

1 **#Define** $EPS = 0.000001$
2 **#Define** $K = 11$
3 **#Define** $L = 5$
4 **procedure** $get\_depth\_bev(d, d\_buffer, color\_img, camera\_intrinsics)$
5 $D = \textbf{cvt\_d2D}(d, intrinsics)$ ▷ see eq. 9.8
6 $pixel\_labels = \textbf{get\_pixel\_labels}(D, color\_img, camera\_intrinsics)$
7 $d_f = \textbf{temporal\_filter}(d, d\_buffer, L)$ ▷ see eq. 9.7
8 $d_f = \textbf{lowpass}(d_f, K)$
9 $D_f = \textbf{cvt\_d2D}(d_f, intrinsics)$ ▷ see eq. 9.8
10 $bev = \textbf{compute\_bev}(D_f, pixel\_labels)$
return bev

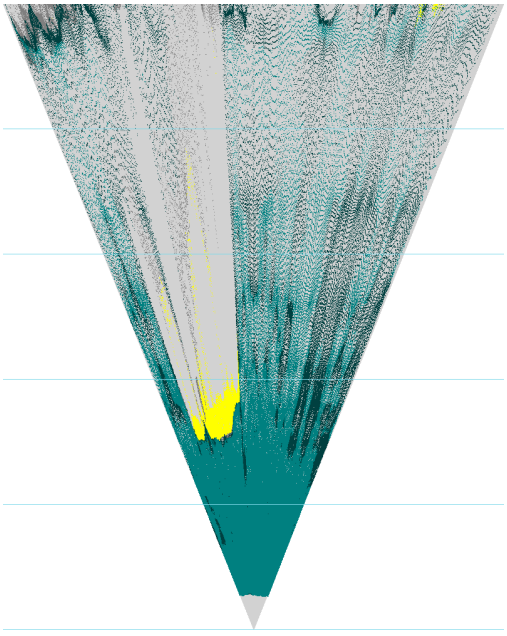


(a) Input color image

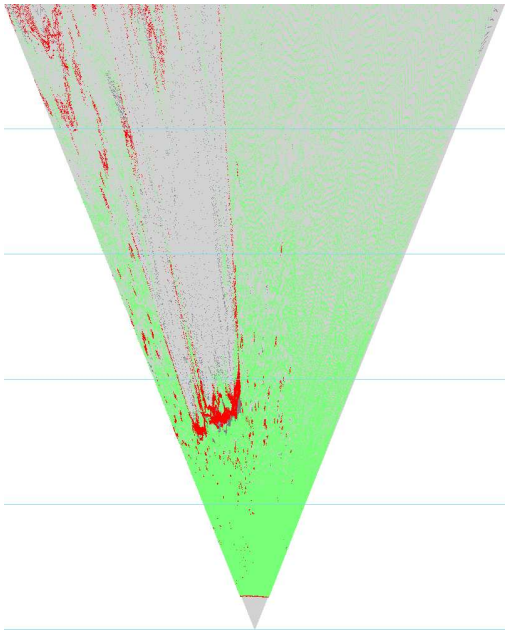
(b) Fused semantic mask



(c) Upright-horizontal labeled mask



(d) BEV with semantic labeling



(e) BEV with upright-horizontal labeling

Figure 9.3: The BEV for different pixel labeling techniques.

Part IV

Ego-motion Estimation

CHAPTER 10

The Ego-motion Estimation sub-system: an overview

Contents

The ego-motion estimation sub-system is mainly responsible for the estimation of pose-change between two image frames using the keypoints-based pose estimation approach. In this chapter, I will give an overview of the architecture of the ego-motion estimation sub-system.

Figure 10.1 shows the simplified architecture of the system in which the *keypoints detector* finds the keypoints all over the image and uses the masks to filter out the non-interested region, whereas the *correspondence estimator* finds the optimized correspondences in three steps: (1) pose change pre-estimation, (2) correspondence prediction, and (3) correspondence optimization. Finally, the correspondences and the pre-estimated pose are used to find the rotation first followed by the optimized pose change. There is another features-based ego-motion estimation module that is being used only to initialize the states (pose) of the system.

Figure 10.1 shows only the main modules and sub-modules of the ego-motion estimator sub-system. There are also some hidden modules (not shown in figure 10.1) that aid the ego-motion estimation. These include the semantic label finder, disparity data filtering, the *keypoints trajectory tracking, etc.* The simple as it seems, the actual implementation is more dense and complex. The modules are arranged in a linear fashion (see fig. 10.1) only for understanding but in actual implementation, they are interconnected.

Algorithm 7 shows the ego-motion sub-system from the implementation point of view. Each module and its components are briefly explained in the following sections.

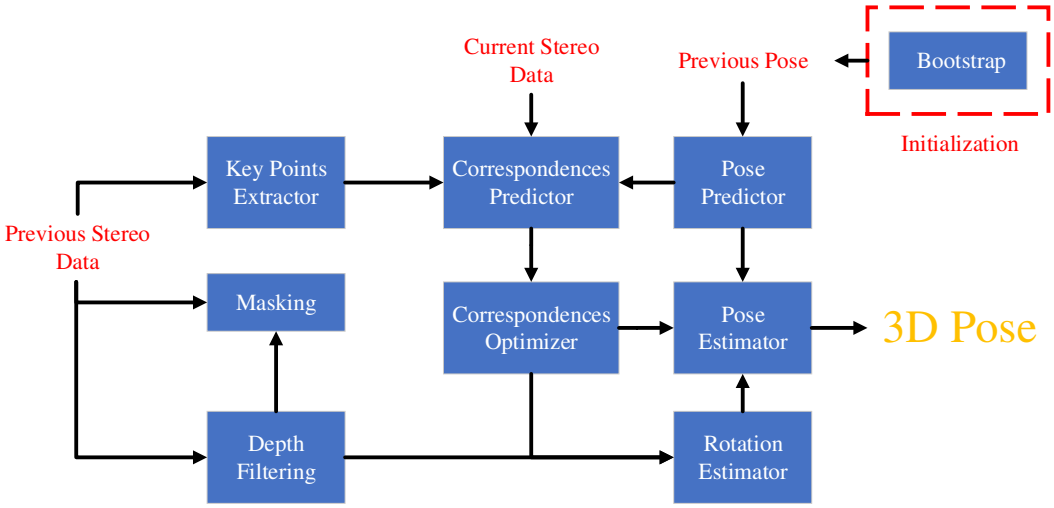


Figure 10.1: Architecture of the ego-motion estimation system.

Algorithm 7 Ego-motion estimation: frame to frame motion

1 $stereo\_data_{t=0} =$ **get_next_stereo_data**()
2 $kps_{t=-1} = None$
3 **for** $t = 1 \rightarrow t = \infty$ **do**
4 $\quad stereo\_data_t =$ **get_next_stereo_data**()
5 $\quad$ **if** $t == 1$ **then**
6 $\quad\quad pose_{t-2}^{t-1} =$ **bootstrap**$(stereo\_data_{t-1}, stereo\_data_t)$
7 $\quad labels_{t-1} =$ **get_semantic_labels**$(stereo\_data_{t-1})$
8 $\quad [kps_{t-2}, kps_{t-1}] \leftarrow corr_{t-1}$
9 $\quad kps_{t-1} =$ **get_kps**$(stereo\_data_{t-1}, stereo\_data_t, labels_{t-1}, kps_{t-1}, pose_{t-2}^{t-1})$
10 $\quad pred\_pose_{t-1}^{t}, corr_t, scores_t =$
$\quad\quad$ **get_correspondences**$(kps_{t-1}, pose_{t-2}^{t-1}, stereo\_data_{t-1}, stereo\_data_t)$
11 $\quad pose_{t-1}^{t} =$ **get_pose**$(corr_t, scores_t, pred\_pose_{t-1}^{t})$
12 $\quad$ **yield** $pose_{t-1}^{t}$

10.1 Get stereo data

The function *get_next_stereo_data* (see step 4 in algorithm 7) collects the stereo images from the camera or the stereo images dataset. Besides the left and right images of the stereo camera, it also returns the disparity image and the depth image. If the camera or the dataset doesn't provide the disparity/depth data then it is being computed first and then returned.

Algorithm 8 Bootstrapping of the pose change

procedure $bootstrap(streo\_data_{t-1}, stereo\_data_t)$
  $init\_pose = I_d$ ▷ I_d is 4×4 Identity matrix
  $init\_rot = None$
  $kps_{t-1}, descriptor_{t-1} =$ **get_kp_n_descriptors**$(streo\_data_{t-1})$
  $kps_t, descriptor_t =$ **get_kp_n_descriptors**$(streo\_data_t)$
  $corr =$ **feature_matching**$(kps_{t-1}, descriptor_{t-1}, kps_t, descriptor_t)$
  $pose^t_{t-1} =$ **pose_estimator**$(corr, stereo\_data_{t-1}, init\_pose, init\_rot)$
  return $pose^t_{t-1}$

10.2 Initialization and bootstrapping of the ego-motion subsystem

This module is run only once during the whole run-cycle of the ego-motion sub-system. It takes the stereo data *stereo_data* at timestep $t = 0$ and $t = 1$ only and performs the features-based ego-motion estimation (see section 15.1, p.132). It extracts the keypoints and their descriptors from both of the images and then matches their descriptors to find the correspondences. After that, it estimates the pose change between two frames using the Perspective-n-Point (PnP) (see section 15.2, p.133). It also assumes the initial pose *init_pose* between the frames as an identity I_d (no pose change) and the initial rotation *init_rot* between them to be zero.

10.3 Keypoints Detector

The keypoint detector module generates the keypoints in an image that are enough in number, widely distributed, and easily trackable. I used the Good Features to Track (GFTT) keypoints and applied different masking strategies to keep control of the location, quality, and distribution of the keypoints (see chapter 12, p.85). The masking methodology is the backbone of the module that creates a mask and tells the keypoint detector whether it should find the keypoint at some pixel or not. For example, the semantic labels (see section 5.2) can also be used as a mask such that the keypoint detector can avoid finding keypoints on fellow boats or on the sea.

The keypoints from the previous timestep that are valid (keypoints with valid correspondences in the current timestep) can be reused while generating the new keypoints (see Fig. 10.2).

10.4 Correspondences estimator

This module estimates the correspondences of all the keypoints in three steps. First, it pre-estimates the pose change between the frames using the previous pose and other advanced techniques (see chapter 11, p.79) and then uses this information to predict the location of the keypoints from the previous frame to the next frame (see section 13.1, p.92). Finally, the correspondences are optimized using a correspondence matcher algorithm (see section 13.2, p.93).

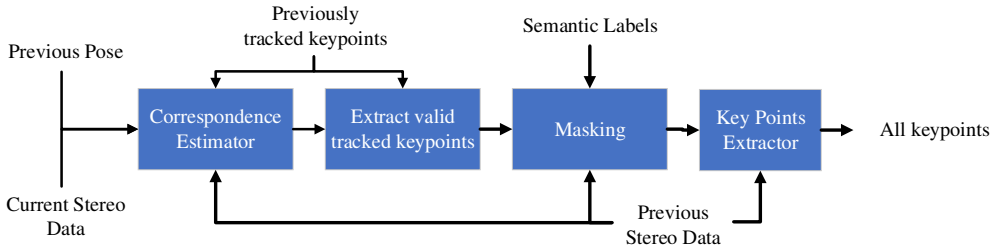


Figure 10.2: Architecture of the keypoints detector module.

Algorithm 9 Keypoints detection using masks

procedure $get\_kps(stereo\_data_{t-1}, stereo\_data_t, semnatic\_labels_{t-1}, kps_{t-1}, pose^t_{t-1})$
  $pred\_pose, corr, scores =$
    get_correspondences$(kps_{t-1}, pose^t_{t-1}, stereo\_data_{t-1}, stereo\_data_t)$
  $kps_{t-1} =$ **filter_valid_kps**$(kps_{t-1}, corr, scores)$
  $masks =$ **generate_masks**$(stereo\_data_{t-1}, semnatic\_labels_{t-1}, kps_{t-1})$
  $new\_kps_{t-1} =$ **generate_keypoints**$(masks, stereo\_data_{t-1})$
  $kps\_t - 1 =$ **merge_kps**$(kps_{t-1}, new\_kps_{t-1})$
  return kps_{t-1}

10.5 Pose change estimator

This module computes the pose change between two frames using the correspondences. It first takes the correspondences from the far-away areas to compute the rotational angles (see chapter 14, p.97) and then uses these rotational angles along with the translation estimates from the pre-estimated pose change to initialize the pose change estimation method. The pose change estimation method uses the keypoints' correspondences and the initialized pose change to estimate the final pose change between two frames using the Perspective-n-Point (PnP) method (see section 15.2, p.133).

Algorithm 10 Correspondence Estimation

procedure $get\_correspondences(kps_{t-1}, pose^t_{t-1}, stereo\_data_{t-1}, stereo\_data_t)$
  $pred\_pose^t_{t-1} =$ **pose_predictor**$(pose^t_{t-1})$
  $kps_t =$ **corr_predictor**$(kps_{t-1}, pred\_pose^t_{t-1}, stereo\_data_{t-1})$
  $corr_t, score_t =$ **corr_optimizer**$(kps_{t-1}, kps_t, stereo\_data_{t-1}, stereo\_data_t)$
  return $pred\_pose^t_{t-1}, corr_t, score_t$

Algorithm 11 Pose change estimator module

1 **procedure** $get\_pose(corr_t, pose^t_{t-1}, stereo\_data_{t-1})$
2 $far\_corr =$ **filter_far_corr**$(corr_t, stereo\_data_{t-1}$
3 $rotation =$ **rotation_estimator**(far_corr)
4 $pose^t_{t-1} =$ **pose_estimator**$(corr_t, pose^t_{t-1}, rotation)$
return $pose^t_{t-1}$

CHAPTER 11

Prediction of the new pose

Contents

The ego-motion estimation sub-system is a keypoint-to-keypoint correspondence-based (also denoted as keypoints-based) pose change estimation system. In the keypoints-based system, the keypoints are detected in one frame and their correspondences are estimated in another frame and based on the correspondences, the pose change is estimated. The accuracy of the estimated pose is highly influenced by the quality of the correspondence and the depth estimates. If the correspondences are estimated at sub-pixel accuracy then the estimated pose is very much accurate and if there is an *outlier* in the set of the correspondences, then it is possible that the estimated pose change may be way far from reality. Avoiding the outliers at all costs, while striving for sub-pixel accuracy in the keypoints-based system is nice to have for accurate pose estimates, but the precision of the resulting relative pose estimate can also be obtained by using many correspondences. But it is important to note that increasing the number of correspondences will in general not help against the effect of outliers.

The outliers are very dangerous and can have catastrophic effects on the system. A loss of precision in the range of fractions of pixels or some 1-2 pixels is unfortunate during the correspondence estimation, as it reduces the precision of the motion estimate, but the outliers can really kill the whole estimation. So the primary goal in correspondence estimation is to avoid outliers. This is done by trying to make very good predictions (first of the 3D motion, and based on this: on the 2D correspondences of all keypoints), and if this is accomplished then reducing the search range for the correspondences as much as possible.

In the later chapter (see chapter 13, p.91), I will talk about the correspondence estimator module which needs the estimate of the pose change between two frames to find the correspondences. This estimate of the pose change is a *pre-estimate* (different from the final optimized pose) and used by the correspondence detector module. As mentioned before, a good pose change

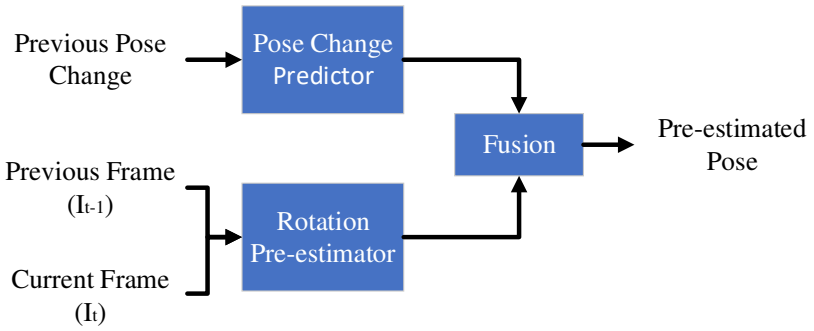


Figure 11.1: Flow chart of the pose change pre-estimator module

pre-estimate allows us to reduce the search range of the correspondence estimator, and this is good for avoiding false correspondences. The importance of the pose change pre-estimate stands out when sudden changes in the camera motion occur, for instance when entering a sharp turn. In this chapter, I will discuss the strategies and methodology to pre-estimate the pose change.

When the new frame (I_t) arrives, the correspondence estimator module intends to predict the new position of a keypoint from frame I_{t-1} on the frame I_t, and for this, it needs two kinds of information: the visual depth of the 3D point corresponding to the regarded keypoint, and the pose change of the camera (translation and rotation). However, it is just this pose change T_{t-1}^{t} that I intend to pre-estimate. As we are in the fortunate situation of dealing with stereo data, I already have the depth estimate from the stereo-matching algorithm. In other words: in order to do correspondence prediction, I need either a *pose change predictor* for the new pose change (typically based on the sequence of already computed poses and a dynamic model of ego-motion) or a coarse *pose change pre-estimator* that preferably should not be based on using keypoints. I intend to use both these approaches in combination, using a rotation pre-estimation and a pose change prediction, where the translation component is the dominantly used part followed by their data fusion.

Figure 11.1 shows the flow of data and the main building blocks of the pose pre-estimator module. The first sub-module *pose change predictor* predicts the pose change given the previous pose change and the second sub-module *rotation pre-estimator* estimates the rotational angles given two image frames.

11.1 Pose change prediction

The pose change information can be predicted up to a certain extent with the help of the dynamic model of ego-motion and such dynamic model-based predictor can also be observed as one of the main components of the predictive filters, such as the Kalman Filter ([26]), Extended Kalman Filter, Particle Filter ([20]), etc. I can not use such predictive filters directly as they also have a correction step in which they update the state (pose) with the help of the measurements, however, the combination of the pose change predictor based on the dynamic model and the pose change correction using the pose change estimation can be regarded as a special case of Kalman filtering. But the main ingredient of the pose change predictor is the dynamic model and the identification of the dynamic model is itself a wide domain that is outside the scope of

this project. Therefore, I decided to explore other predictors that can be implemented without the knowledge of the dynamic model.

11.1.1 Statistics-based motion predictor

[6] proposed a linear predictor for the motion parameters based on the statistics of the ground truth from the KITTI dataset. Let $\vec{p}[n]$ be the motion parameter vector that defines the motion between the timestep n and *n-1*, then they built a linear motion predictor that can predict the relative motion parameter vector $\hat{\vec{p}}[n+1]$ between timestep *n+1* and n by learning the motion statistics.

$$\hat{\vec{p}}[n+1] \stackrel{\text{def}}{=} \mathbf{E}\left[\vec{p}[n+1] \mid \vec{p}[n]\right] \tag{11.1}$$

The learned motion parameters were the mean of the motion parameter $\vec{m}_n$ and $\vec{m}_{n+1}$ at timestep n and *n+1* respectively and the cross-correlation $\mathbf{C}_{n+1,n}$ and auto-correlation covariance $\mathbf{C}_{n,n}$ matrices. The learned auto-correlation covariance matrix $\mathbf{C}_{n,n}$ can tell us the intra-vector (motion parameter vector) correlations and the cross-correlation covariance matrix $\mathbf{C}_{n+1,n}$ can tell us if any of the motion parameters can be predicted from their temporally preceding values. They considered $\vec{p}[n]$ as a stationary process (independent of time) such that $\vec{m}_n = \vec{m}_{n+1}$. The predictor is a *linear* predictor and therefore can be described by a linear prediction equation that uses a *predictor matrix* $\mathbf{A}$.

$$\hat{\vec{p}}[n+1] = \mathbf{A} \cdot (\vec{p}[n] - \vec{m}_n) + \vec{m}_n \tag{11.2}$$

where the predictor matrix $\mathbf{A}$ is

$$\mathbf{A} \stackrel{\text{def}}{=} \mathbf{C}_{n+1,n} \cdot \mathbf{C}_{n,n}^{-1} \tag{11.3}$$

The statistics-based motion model is very useful to reject the motion outliers that can arise from sensor (IMU) noise, vibration of the ego-vehicle, etc.. and can be used to predict the next pose change as the dynamic characteristics of any vehicle can be represented by the statistics of the ego-motion data. Despite the model's usefulness, I didn't implement the same in my project because of the need for ego-motion data, which is hardly available for maritime vehicles. Also, the ego-motion data if available can be used to model only for the vehicle on which it was recorded to obtain the best results but this is a general problem of all the model-based approaches.

11.1.2 Previous pose change based motion predictor

It is the simplest motion predictor among all of the motion predictors as it assumes the ego-vehicle has constant velocity and no acceleration since the last timestep and therefore, the current pose change is equal to the last pose change.

Let $\mathbf{R}_{t-2}^{t-1}$ and $\vec{t}_{t-2}^{t-1}$ be the rotation matrix and the position vector between the timesteps *t-1* and *t-2* respectively, then the pose change $\mathbf{T}_{t-2}^{t-1}$ can be composed of the rotation matrix $\mathbf{R}_{t-2}^{t-1}$ and the translation vector $\vec{t}_{t-2}^{t-1}$.

$$\mathbf{T}_{t-2}^{t-1} = \begin{bmatrix} \mathbf{R}_{t-2}^{t-1} & \vec{t}_{t-2}^{t-1} \\ 0_{3\times1} & 1_{1\times1} \end{bmatrix} \tag{11.4}$$

Similarly, let $\hat{\mathbf{R}}_{t-1}^{t}$ and $\hat{\vec{t}}_{t-1}^{t}$ be the predicted rotation matrix and the predicted position vector between the timesteps t and *t-1* respectively, then the predicted pose change $\hat{\mathbf{T}}_{t-1}^{t}$ can be computed from the last pose change.

$$\hat{\mathbf{T}}_{t-1}^{t} = \mathbf{T}_{t-2}^{t-1} \tag{11.5}$$

such that,

$$\begin{aligned}\hat{\mathbf{R}}_{t-1}^{t} &= \mathbf{R}_{t-2}^{t-1} \\ \hat{\vec{t}}_{t-1}^{t} &= \vec{t}_{t-2}^{t-1}\end{aligned} \tag{11.6}$$

This approach has huge limitations when the ego-vehicle undergoes sharp maneuvers. When the vehicle accelerates, the previous pose change underestimates the new pose change and overestimates when the vehicle de-accelerates. Due to the duration time for this project, I couldn't explore more options and decided to use this pose change predictor.

11.2 Rotation pre-estimation

As we know, the pose consists of two types of motion: rotation and translation. When the camera undergoes some motion, the pixels in the image displaces from their original position. The effect of translation on the displacement of the image points diminishes if they are far enough from the camera as the 2D displacement is inversely proportional to the depth of the corresponding 3D point and therefore, this displacement is caused solely by rotation motion. In other words, if the 2D displacement of the far-away points in the image plane is known then it can be used to estimate the rotation assuming that the translation has no or negligible influence on this 2D displacement.

It should be noted that the sole purpose of the rotation pre-estimator is to have a coarse rotation estimate that can be used by the correspondence predictor. It implies that the keypoint's correspondences are not available to the pre-estimator and therefore, it should use other than the keypoints' correspondences. In chapter 14, I will discuss the rotation estimation from far-away areas in detail. I will also introduce some methods to compute one or more rotational angles based on the far-away areas or the far-away keypoints.

11.3 Fusion of pose change prediction and rotation pre-estimation

In the previous sections, I talked about a very simple pose change predictor that uses the previous pose change as the predicted pose change and the rotation pre-estimator that gives a coarse estimate of one or more rotational angles based on the two image frames. The full or part of the pose information from these two methods should be fused for a final pre-estimation of the pose change such that it can be used by the correspondence detector.

The predicted pose change $\hat{\mathbf{T}}$ can be decomposed into the translation vector $\hat{\vec{t}}$ and the rotation matrix $\hat{\mathbf{R}}$.

$$\hat{\mathbf{T}} = \begin{bmatrix} \hat{\mathbf{R}} & \hat{\vec{t}} \\ 0_{1\times 3} & 1_{1\times 1} \end{bmatrix} \tag{11.7}$$

The rotation matrix is further decomposed into roll, pitch, and yaw Euler angles using the algorithm 23 (see appendix A.4, p.174). Let us assume that the rotation pre-estimator estimates only the yaw angle, then this yaw angle is replaced with the yaw angle from the pose change predictor and then the modified Euler angles are converted back into the rotation matrix followed by the formation of the transformation matrix (see algorithm 12). Similarly, if more than one rotational angles are computed, then they are replaced with the predicted rotational angles and the remaining steps remain the same. These simplifications neglect the fact that for many real

vehicles, rotation, and translation are to some degree coupled to each other. This coupling is represented in Bradler's prediction model ([6]).

It should be noted that due to the limited time of the project, I managed to explore the different methods for the rotation pre-estimator but I couldn't manage to fine-tune or analyze them properly to actually use them in the module. Therefore, I used only the pose change predictor to generate the final results, however, a detailed description of the rotation estimation from far-away areas is given in chapter 14 for future work.

Algorithm 12 Fusion of pose prediction and rotation pre-estimation

1 **procedure** $fuse\_pose(prev\_pose,\ prev\_img,\ prev\_depth\_img,\ curr\_img,\ curr\_depth\_img)$

2 $pred\_pose = \textbf{pose\_predictor}(prev\_pose)$

3 $yaw\_pre\_est =$
$\textbf{rotation\_estimator}(prev\_img, prev\_depth\_img, curr\_img, curr\_depth\_img)$

4 $[\hat{\mathbf{R}}, \hat{\vec{t}}] = \textbf{decompose}(pred\_pose)$

5 $[r, p, y] = \textbf{rotation2euler}(\hat{\mathbf{R}})$ ▷ see alg. 23.

6 $fused\_rot = \textbf{euler2rotation}(r, p, yaw\_pre\_est)$ ▷ see eq. A.9.

7 $fused\_pose = \textbf{compose}(fused\_rot, \hat{\vec{t}})$
return $fused\_pose$

CHAPTER 12

Keypoint Generation using Masking

Contents

A reliable and robust Keypoints Detector (KptDet) module acts as a foundation block for any keypoints based vision pipeline. The KptDet module should be able to identify the pixels (keypoints) from the image that have unique properties and differentiate them from the other pixels such that they can be robustly tracked in subsequent images. An individual pixel doesn't contain enough information, therefore, the neighborhood around the pixel is used instead. The neighborhood is a sub-image or a patch of pixels centered at the pixel under consideration. The key feature of any keypoint is that it should have enough texture around it otherwise the displacement of homogeneous surfaces from one image to another can't be estimated using a keypoints-based approach.

In the past, many keypoint extractors have been proposed. [31] proposed a scale-invariant keypoint extractor called SIFT. It is very robust but computationally expensive. Similarly, other keypoints extractors such as SURF ([3]) and ORB ([39]) fall under the most popular keypoints extractor category. The most common properties of these feature extractors were to find the keypoint and generate a unique description of the keypoint based on the structure of the neighborhood. This description is used to match the keypoint against the keypoints in the next image. In this project, I have decided to use Good Features to Track (GFTT) ([43]) to identify the keypoints as in [15]. The GFTT is an extension of the Harris corner detector.

I used the OpenCV's implementation for GFTT[1] to find the keypoints in the image. The implementation is based on the original paper ([43]). The constraints offered by the function to filter the keypoints are the following.

[1]The link for the tutorial is given here.

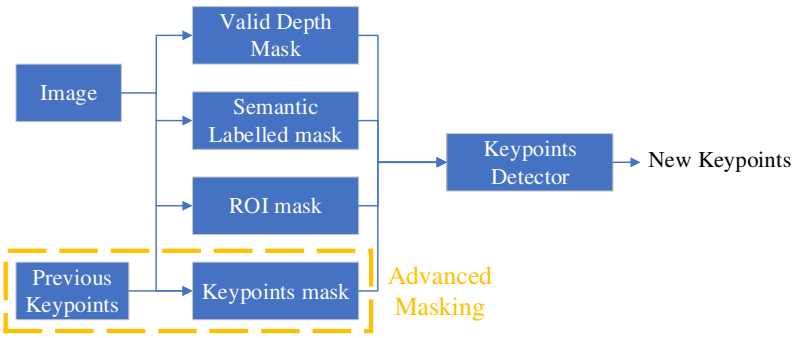


Figure 12.1: The flowchart for the KptDet algorithm.

- **Maximum Corners:** This parameter defines the maximum number of keypoints to be extracted. If there are more corners than the threshold, the strongest of them have been kept.
- **Quality Level:** It filters the corners according to their quality measure relative to the keypoint with the best quality. For example, if the best keypoint has a quality measure of 1000 then with a quality level of 0.15, all the keypoints with a quality measure less than 150 are not considered.
- **Minimum Euclidean Distance:** The minimum Euclidean distance between any two keypoints[2].
- **Masking:** It is possible to mask out the region from the image that is not interested for the keypoints generation using a binary mask of **True** and **False**. The pixels that have **False** in the mask are rejected for the keypoints.

12.1 Keypoint masking

In an outdoor scene, a considerable part of the image is always covered with the sky, and specifically in maritime scenes, water, and sky cover the most part of the image. In an ideal scenario for the computation of ego-motion from the keypoints matching, the keypoints should be generated in the whole image except for the textureless regions and the dynamic objects in the image. The homogeneity can arise from the walls of the building, sky, etc., and sea, clouds, other objects in motion, etc. covering the dynamic objects of the scene. It is critical to differentiate between the homogeneous and textured region and the static and dynamic part of the scene. I have attempted to resolve this approach using the masking feature of the KptDet module.

In the following sections, I will talk about the different masking strategies that have been used.

12.1.1 Valid depth mask

The depth of each keypoint has been taken into account during the keypoint pose prediction, rotation estimation using far keypoints, and the ego-motion estimation. The rejection of the

[2]I tried to understand the source code of the function in OpenCV and I found out that the function compares the distance of the found keypoints with other keypoints to maintain the distance.

keypoints with invalid depth measures at the earlier stage of the processing increases the accuracy as well as reduces the chance of failure of the system. I created a mask using the disparity image of the stereo-pair. In the mask, all the pixels that have no disparity data due to occlusion, etc, are marked **False** and the mask is created.

12.1.2 Region of Interest (ROI) mask

In some peculiar setups of the camera system or the environment, it is possible to have a consistent region or space in the image that should be disregarded before extracting the keypoints. For example, if the deck of the ship is in the view of the camera, then the pose estimated from the keypoints on the deck is always null and can force the pose estimator to think that the ego-vehicle is stationary. Similarly, if the projection of the sky on the image is consistent then it can be disregarded completely by masking it out. The knowledge of the environment can be used very efficiently while extracting the keypoints. Taking these points into consideration, I have provided the feature to generate a static mask where pixels are marked **False** for those uninterested regions.

12.1.3 Semantic labelled mask

As we know, the motion of any object is relative to some reference frame and the ego-pose estimator shouldn't confuse itself with the motion of other dynamic objects in the environment. For example, in a harbor scene, the other dynamic objects are the fellow ships and boats and if the keypoints lie on them then the pose estimated between two consecutive frames will be biased towards the motion of the ego-vehicle with respect to the other vehicles instead of the inertial reference frame.

The identification of the other dynamic bodies can be done using a semantic segmentation mask generated from the machine learning techniques. In chapter 7, I mentioned multiple strategies to segment the water plane and the fellow boats. I used the fused mask (see algorithm 5, p.52) and convert it into a binary mask such that the labels that don't correspond to sea, water, or boat are marked **False** in the mask or **True** otherwise[3].

12.1.4 Advanced masking: Keypoints mask

The tracking of keypoints across multiple frames is an important way to improve the precision of the ego-motion estimation. Some keypoints, in particular those which are in the far distance and in the motion direction of the vehicle, are visible for a very long time, and establishing the correspondence between a keypoint in frame N and a keypoint in frame $N + K$ is very valuable for stabilizing the ego-motion, in particular, the estimates of the rotation, if the frame difference K is large.

The actual design idea of the ego-motion subsystem is to track keypoints from frame to frame as long as they are visible. This is good for having correspondences over substantial motions, and this improves the precision of the motion estimation. However, due to the limited time, I could only prepare the keypoint tracking system for this capability, but could not fully implement it. In the current system, I am re-using the keypoints that have been tracked from the image at timestep $t-1$ into the image at timestep t and trying to track it into the image at timestep $t+1$. In future work, the trajectory of the keypoints shall be used to post-optimize the ego motion and the trajectory of the ego vehicle. In this section, I will discuss how to deal with the keypoints

[3]The semantically labeled mask is available only for maritime scene.

coming from the previous frames such as the clusters between the previously tracked keypoints and the new keypoints can be avoided.

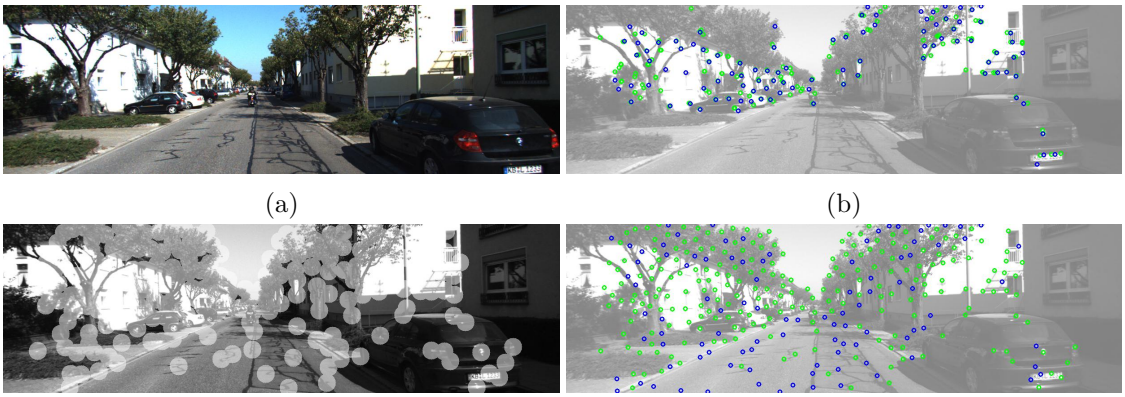
The OpenCV's implementation of the GFTT provides an argument called *minDistance* to limit the minimum distance between any two keypoints. As stated earlier, this can be used to avoid the clusters of keypoints. However, it can not avoid the generation of new keypoints near previously tracked keypoints. It can be possible by creating a mask for the previously tracked keypoints. For each previously tracked keypoint, a circular patch of radius equal to the *minDistance* around the keypoint is assigned **False** in the mask. With this mask, no new keypoint is generated near to the previous tracked keypoint and the minimum distance between keypoints can be maintained.

The keypoints mask is an advanced feature of the module and can be used only in the presence of previously tracked keypoints. Figure 12.2 shows the effect of the keypoint mask on the distribution of the keypoints[4]. In the presence of previously tracked keypoints (represented as green in color in fig. 12.2a and fig. 12.2b), the KptDet module found new keypoints (represented as blue in color) near to previously tracked keypoints in the absence of the keypoints mask and resulted into a cluster of keypoints. It should be noted that the new keypoints are themselves not clustered together because of the *minDistance* between the new keypoints. When the keypoints mask (see fig. 12.2c) was introduced, the KptDet module was forced to look for new keypoints far from the previously tracked keypoints (see fig. 12.2d) and distributed the keypoints all over the image.

12.1.5 Fusion of the masks

The different kinds of masks discussed above can be fused together to form a single boolean mask. There are different complex strategies that can be adapted to fuse them, however, I opted for a simple one and fussed all of them using a logical *and* operation. If the pixel from any of the masks is set to **False**, then the corresponding pixel in the resultant mask is **False**. The fused boolean mask needs to be mapped to an 8-bit unsigned integer value such that it can be accepted

[4]It should be noted that the parameters used to generate these results are only temporary and have no influence on the final module.



(a) (b)

(c) (d)

Figure 12.2: The extraction of keypoints using the keypoints mask.

Table 12.1: Parameters for KptDet module

Parameters	code name	Values
Maximum total keypoints	*max_total_kps*	100
Minimum total keypoints	*min_total_kps*	4
Quality level	*quality_level*	0.01
Minimum distance between keypoints	*min_kp_dist*	20 pixels
Valid depth mask	*use_depth_mask*	True
ROI mask	*use_roi_mask*	False
Semantic labelled mask	*use_label_mask*	False
Keypoints mask	*use_keypoint_mask*	True

by the function. The **False** is mapped to **0** and **True** is mapped to **1**. The KptDet module finds the keypoints only for those pixels that have a non-zero value in the resultant mask.

12.2 Implementation of Keypoint Detector

The main and the only role of the KptDet module is to provide good quality of keypoints. The maximum number of keypoints should be limited to check the computational usage and the minimum number of keypoints should be enough for the pose computation. I decided to choose the maximum number of keypoints to be *max_total_kps* to begin with and analyze the distribution of the keypoints along the sequence and the minimum number of keypoints to be *min_total_kps* as there should be at least 4 keypoints to compute the rotation from far-away points ([2]). It should be noted that if I allow the KptDet module to generate the maximum number of keypoints for each frame then the total number of keypoints (= previously tracked keypoints + newly generated keypoints) may exceed the maximum limit *max_total_kps*. Similarly, if I have enough previously tracked keypoints, then the minimum number of new keypoints will be different as well. Let *prev_track_kps* be the number of keypoints that have been tracked from the previous frames and can be tracked successfully using the correspondence estimator (see chapter 13, p.91) then the maximum and minimum limit for the new keypoints can be computed using the following equations.

$$\begin{aligned} max\_new\_kps &= max(0, max\_total\_kps - prev\_track\_kps) \\ min\_new\_kps &= min(min\_total\_kps, max(min\_total\_kps - prev\_track\_kps, 0)) \end{aligned} \tag{12.1}$$

In case of no keypoint tracking ($prev\_track\_kps = 0$), the maximum and minimum number of keypoints are *max_new_kps* and *min_new_kps* respectively.

12.3 Discussion on keypoints generation

The OpenCV's implementation for the GFTT keypoints is very flexible in usage. It is possible to set the maximum number and the relative quality of the keypoints. Also, it is possible to define the interested regions through different masks. It should be noted that the different masks that I suggested before can vary depending on the usage. The valid depth mask and the keypoint mask require no knowledge of the environment and can be used directly. For the time being, I have restrained myself to use ROI mask because it requires additional knowledge of the environment and limited the use of the semantically labeled mask for the maritime sequence only.

I ran the KptDet module only on the KITTI data from seq. 00 to seq. 10 to check if it is able to find the *max_total_kps* for every frame irrespective of any masking and tracking. I found out that the success rate for this module was 100 percent and there was no frame in those sequences for which the module failed to generate the desired number of points. It implies that the standalone module to generate the keypoints is sufficient and can be coupled with the other modules.

CHAPTER 13

Correspondences Prediction and Optimization

Contents

In the previous chapter, I talked about the generation of the Good Features to Track (GFTT) keypoints from the image. These independent keypoints from the frame I_{t-1} and the frame I_t can not tell us about the motion that occurred between these two frames. To measure the motion between two frames, we need the correspondences of the keypoints from one frame to another. In other words, we need to find where the keypoints appear in the next frame after undergoing some unknown motion and then recover the motion from them. This is called the correspondence problem in which the correspondences of the keypoints from frame I_{t-1} are found in frame I_t. In an ideal case, the corresponding keypoint should be found at a sub-pixel accuracy, however, due to the change of illumination, sensor noise, and limited camera resolution, there is always a remaining measurement error. This error is typically in the sub-pixel range, or larger, depending on the chosen method.

In the introductory chapter (see section 3.2.2, p.20), we saw that there could be multiple approaches to find the correspondences, and the most popular among these approaches are the Block Matching (BM), differential matching, and the phase correlation (PhC) matching method. The BM is very simple to use but limited to pixel-level accuracy. The differential matching algorithm requires a number of iterations and good initial correspondence estimates to converge to the optimal result and the classical phase correlation-based method assumes the homogeneous motion in the regarded patch which is not true in the complex motion.

I will discuss the Correspondence Estimator (CorrEst) module in this chapter that consists of a correspondence predictor followed by a correspondence optimizer which is based on Lukas-Kanade (LK) differential matcher because of its sub-pixel accuracy.

13.0.1 Pose pre-estimator: Rotation pre-estimation

When the ego-vehicle is doing smooth maneuvers, the prediction of the correspondences in the next frame is very close to the reality, however, during sharp turns, the correspondences start to undershoot or overshoot and as a result, the correspondence optimizer may fail to find the correspondences. Taking this potential issue into consideration, I attempted to pre-estimate the rotation motion. It should be noted that the yaw is the most dominant rotation motion for any maritime vessel or ground vehicle, therefore, I tried to estimate the yaw motion between the previous frame and the new frame with an assumption that roll and pitch are zero.

13.1 Correspondence predictor

The performance of the LK matcher is tightly coupled with the initialization of the keypoints' correspondences. The simplest approach is to use the pixel coordinates of the keypoints as the initial locations for the correspondences. This approach may work for the farthest keypoints under a small ego-motion but for the nearest keypoints, the 2D displacements are huge and can lead to false or no correspondences. This issue can be resolved by using a large search window by the matcher, however, the accuracy in the location of the correspondences are reduced significantly as it increases the probability to estimate incorrect correspondences because the image patches look similar. To overcome this challenge, I decided to predict the location of the keypoints from the previous frame into the current frame using a correspondence predictor module.

Let $[x'_1, x'_2]^T$ be the keypoint in pixel coordinate, d be the depth of the keypoint depth in the frame I_{t-1}, and $[X_1, X_2, X_3]^T$ be the corresponding 3D coordinate of the keypoint w.r.t the Camera Coordinate Frame (CCF) at timestep $t-1$ then the keypoint can be projected from the image space to the 3D space.

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = d \begin{bmatrix} \frac{(x'_1 - c_x)}{f_x} \\ \frac{(x'_2 - c_y)}{f_y} \\ 1 \end{bmatrix} \tag{13.1}$$

Let $[Y_1, Y_2, Y_3]^T$ be the 3D coordinate of the same keypoint $[x'_1, x'_2]^T$ w.r.t the CCF at timestep t, $\hat{\mathbf{R}}$ and $\hat{\vec{t}}$ be the rotation matrix and translation vector respectively computed by the pose change pre-estimator (see chapter 11, p.79), then the 3D point $[X_1, X_2, X_3]^T$ can be transformed into the $[Y_1, Y_2, Y_3]^T$.

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \rightarrow \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \hat{\mathbf{R}} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \hat{\vec{t}} \tag{13.2}$$

Finally, this 3D point $[Y_1, Y_2, Y_3]^T$ can be projected back on the frame I_t.

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \frac{1}{Y_3} \begin{bmatrix} f_x Y_1 + c_x Y_3 \\ f_y Y_2 + c_y Y_3 \end{bmatrix} \tag{13.3}$$

The keypoints from frame I_{t-1} can be projected on frame I_t using eq. 13.1 to eq. 13.3. The keypoint predictions give a good initial estimate for the LK matcher.

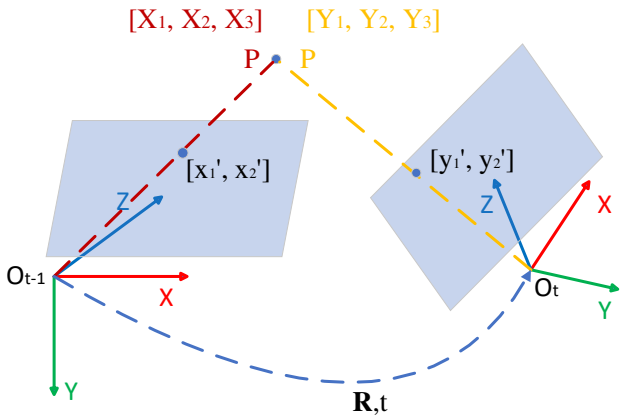


Figure 13.1: The prediction of keypoint $[x_1', x_2']^T$ from frame I_{t-1} to frame I_t.

Algorithm 13 Correspondence Predictor

```
procedure pred_corr(kps, kps_depth, pred_pose, camera_intrisics)
    3D_kps ← PIXEL_TO_3D(kps, kps_depth, camera_intrinsics)        ▷ refer eq. 13.1
    3D_trans_kps ← TRANSFORM(3D_kps, pred_pose)                    ▷ refer eq. 13.2
    pred_corr ← 3D_TO_PIXEL(3D_proj_kps, camera_intrisics)         ▷ refer eq. 13.3
    return pred_corr
```

13.2 Correspondences optimization

The correspondence estimate obtained from the keypoint pose predictor can be accurate only if the depth data and relative pose are precisely known, however, it is not true due to the limited accuracy in-depth and the lack of knowledge of true pose change. The correspondences should be close to the ground truth as much as possible for the pose estimation, therefore, some optimization needs to be done. I used the OpenCV's implementation for pyramidal implementation[1] of the LK matcher ([5]) method to optimize the correspondences from I_{t-1} to I_t. The pyramid implementation can handle the large motion between the frame without increasing the window size. The parameters to tune the function are the following.

- **Search window size:** It is the search window size at each pyramid level.
- **Number of pyramid levels:** The number of maximum pyramid levels that the algorithm may search in to find the correspondences.
- **Termination criteria:** The function allows two termination criteria. The first one is the maximum number of iterations and the second is the convergence of the displacement vector. If the displacement in the next iteration is below this value then the criteria is fulfilled.

[1]The tutorial for the function is given here.

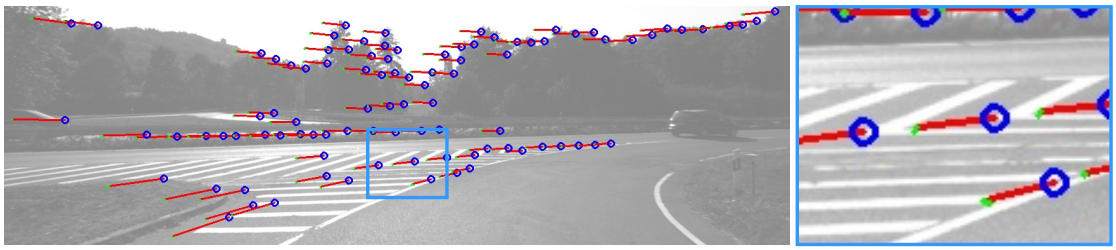


Figure 13.2: Correspondence prediction and optimization on frame pair 12-13 of KITTI seq. 01. The part of the image is magnified and shown on the right. The blue circle shows the keypoints in frame 12 and the other end of the red line is prediction and the end of the green line that stretches from the red line is the final optimized correspondence.

Table 13.1: Initial set of parameters for LK matcher

Parameters	code name	Values
Search window size	lk_win_size	20×20
Pyramid levels	lk_num_pyrs	0
Maximum iterations	lk_itr	5
Minimum displacement (ϵ)	lk_epsilon	0.1
Correspondence error threshold	lk_l1_th	25

It is very crucial to choose the values of the number of pyramid levels and the search window size as these parameters highly influence the accuracy and robustness. During large ego-motion, the pose estimates can be significantly wrong and can lead to the poor initialization of the correspondences. The increase in the window size can provide robustness but at the cost of accuracy. Therefore, a pyramid level can be introduced instead to handle the large motion without compromising the accuracy of the matcher, however, a good correspondence prediction may also eliminate the need for the pyramid approach. Assuming that I have a good correspondence predictor, I decided to choose no pyramid levels (the zero/base pyramid level is the original raw image) and a window size of *lk_win_size*. The correspondences from the LK matcher can't be trusted blindly and therefore, the L1 residual should be monitored and thresholded to filter out the false correspondences. I compared the L1 residual obtained between the patches around the current keypoint and the corresponding keypoint by the OpenCV's function with the *lk_l1_th* threshold. Any correspondence with a residual above this value is disregarded. Table 13.1 lists the initial set of parameters and their values for the LK matcher.

Algorithm 14 Correspondence Optimization

```
1 procedure opt_corr(kps, pred_kps, prev_img, curr_img)
2     lk_params = {lk_num_pyrs, lk_win_size, lk_epsilon, lk_itr}
3     opt_corr, error = LK_MATCHER(kps, pred_kps, prev_img, curr_img, lk_params)
4     if error ≤ lk_l1_th then
5         return opt_corr
6     else
7         return NONE
```

13.3 Discussion on the correspondence estimator

Unlike the KptDet module, I can't run the CorrEst module independently because of the need for the previous pose by the correspondence predictor. I ran the pipeline on different sequences from the KITTI dataset using the parameters defined in table 13.1. Figure 13.2 shows that the correspondence predictor managed to give a good initial estimate and then the correspondence got optimized at sub-pixel level accuracy by the LK matcher. It should be noted that the valid correspondences between any pair of frames can't exceed the maximum number of keypoints i.e. *max_total_kps*.

CHAPTER 14

Rotation from far away areas

Contents

In this chapter, I will extend the discussion from section 11.2 on the frame-to-frame change of the pitch, yaw, and roll angles from the far-away region. I have already mentioned that the rotation and translation motion can be decoupled if we look at the regions in the image that are far from the camera because the 2D displacement of the far-away pixels in the image occurs due to rotation (the translation will have a very little influence) but in this chapter, I will prove the same.

In [2], the authors proposed a methodology to compute the pitch, yaw, and roll based on far-field windows. Inspired by their approach, I decided to explore the same principle of estimating the rotational angles from far-away areas due to the following reasons.

- **Availability of the depth:** In the case of a stereo camera, the reliable farthest points are easy to identify.
- **Decoupling between translation and rotation:** As the point goes to infinity, it can be proved that the 2D motion (i.e. the displacement between corresponding pixel coordinates) becomes independent of the translation motion.
- **Problem simplification:** Having rotation from a far-away region first simplifies the translation estimation problem as instead of solving 6 non-linear equations simultaneously for each degree of freedom, translation can be estimated using only 3 linear equations.
- **Accuracy:** The results shown in [2] are very promising and encouraging.

[2] used roll, pitch, and yaw Euler angles to represent the rotation (the coordinate frame and notations will be explained later in section 14.1.1, p.98) and made a few assumptions such that the motion vector field (= displacement vector field) of far away points is well approximated by

- the yaw leading to horizontal shift (only), and nothing else
- the pitch leading to vertical shift (only), and nothing else
- the roll leading to in-plane rotation by the same angle (only), and nothing else

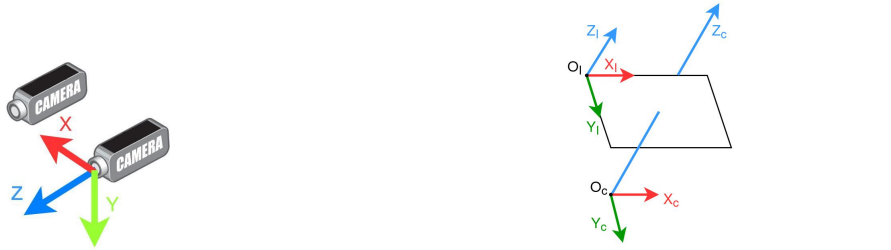
These assumptions are if the image plane was not a plane but a sphere, and it is this deviation of the image plane from being a sphere that causes these deviations from the 2d motion field model defined in [2]. In the following sections, I will derive the 2D displacement equations from the motion of the camera and prove that the 2D motion becomes independent of the translation for far-away areas followed by answering the question "What can be considered as a far-away point to neglect the coupling between translation and rotation?". After this, I will derive the equations for the estimation of rotation angles from the 2D motion and later propose three different methods for the rotation estimation. One of the three methods is the simplified implementation of the algorithm proposed in the [2]. The second method uses the depth estimate which is missing in the previous method and estimates only the yaw angle. Finally, the third method uses the keypoints' correspondences which is completely different from the first two area-based approaches. The proposed algorithms have been tested on the KITTI dataset due to the availability of the ground truth.

14.1 3D Motion Analysis from 2D motion field

In this section, I will discuss how the 3D motion (rotation and translation) affects the displacement between a pair of corresponding pixels. If a camera undergoes a 6-degree of freedom motion (3 for rotation and 3 for translation) then it results in a very complicated expression and becomes very difficult to analyze. To make it comprehensible, I considered each motion independently and studied its effect on the 2D motion field. Before going any further, I have defined the reference frame in which the rotation and translation happen.

14.1.1 Coordinate Frame

I define three coordinate frames; the World Coordinate Frame (WCF), the Camera Coordinate Frame (CCF), and the image coordinate frame. I used the same convention as [19] to define these reference frames. In the CCF, the $z-axis$ points forward, the $y-axis$ points downwards, and the $x-axis$ points rightwards (see Fig.. 14.1a). The WCF is aligned with the camera's



(a) Camera coordinate system (b) Camera and Image coordinate system

Figure 14.1: Coordinate systems for the camera and image plane.

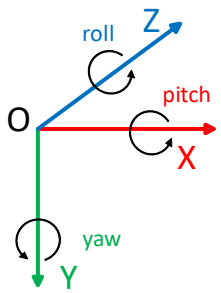


Figure 14.2: The rotational angles w.r.t the CCF.

coordinate system at time $t = 0$. It is a fixed frame and doesn't change with time. The image coordinate frame is located on the top left of the image with $x - axis$ on the rightwards of the image ($= X_I$) and $y - axis$ is downwards ($=Y_I$) (see Fig.. 14.1b). The image plane coordinates are usually used to define the pixel coordinates of the keypoints.

[2] used the roll, pitch, and yaw Euler angles to represent the rotational angles about the $z - axis$, $x - axis$, and $y - axis$ respectively (see Fig. 14.2). I used the same notations for the rotational angles in this project.

14.1.2 2D motion from general motion of camera

In section 13.1, we saw how the keypoint with pixel coordinates $[x_1', x_2']^T$ transformed into the pixel coordinates $[y_1', y_2']^T$ after undergoing some motion. The expressions used in that section are rewritten here for readability.

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = d \begin{bmatrix} \frac{(x_1' - c_x)}{f_x} \\ \frac{(x_2' - c_y)}{f_y} \\ 1 \end{bmatrix} \tag{14.1}$$

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \rightarrow \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \mathbf{R} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \vec{t} \tag{14.2}$$

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \frac{1}{Y_3} \begin{bmatrix} f_x Y_1 + c_x Y_3 \\ f_y Y_2 + c_y Y_3 \end{bmatrix} \tag{14.3}$$

Let Δ be the 2D displacement of the keypoint due to this motion.

$$\Delta = \begin{bmatrix} y_1' \\ y_2' \end{bmatrix} - \begin{bmatrix} x_1' \\ x_2' \end{bmatrix} \tag{14.4}$$

If the coordinates of the keypoint correspondences are given in image coordinates (in meters) instead in the pixels, then the expression for the 2D displacement Δ can be simplified and easy to understand. Let $[x_1, x_2]^T$ be the image coordinates of the keypoint, $[y_1, y_2]^T$ be the image coordinates of the correspondence, and f be the focal length in meters then the above expressions can be simplified.

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} x_1 \frac{d}{f} \\ x_2 \frac{d}{f} \\ d \end{bmatrix} \tag{14.5}$$

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} = \mathbf{R} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \vec{t} \tag{14.6}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} f \frac{Y_1}{Y_3} \\ f \frac{Y_2}{Y_3} \end{bmatrix} \tag{14.7}$$

$$\Delta = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{14.8}$$

In the proposed module, the keypoints are expressed in pixel coordinates, therefore, I focused on the equations that we derived for the pixel coordinates only and not for the image coordinates. Also, in the following sections, I will restrict the $\mathbf{R}$ and $\vec{t}$ to have at most one degree of freedom together.

14.1.3 Rotation about $x - axis$ (Pitch)

Let ϕ be the pitch angle or the rotation about the $x - axis$ in the CCF and when the camera undergoes the pitch ϕ motion, then the equation 14.1 remains the same and the equations 14.2

and 14.3 get simplified as below.

$$\begin{aligned}\begin{bmatrix}Y_1\\Y_2\\Y_3\end{bmatrix} &= \begin{bmatrix}1 & 0 & 0\\0 & \cos\phi & -\sin\phi\\0 & \sin\phi & \cos\phi\end{bmatrix}\begin{bmatrix}X_1\\X_2\\X_3\end{bmatrix} + \begin{bmatrix}0\\0\\0\end{bmatrix}\\ &= d\begin{bmatrix}\dfrac{(x'_1-c_x)}{f_x}\\ \dfrac{(x'_2-c_y)}{f_y}\cos\phi - \sin\phi\\ \dfrac{(x'_2-c_y)}{f_y}\sin\phi + \cos\phi\end{bmatrix}\end{aligned} \tag{14.9}$$

$$\begin{aligned}\begin{bmatrix}y'_1\\y'_2\end{bmatrix} &= \frac{1}{Y_3}\begin{bmatrix}f_xY_1 + c_xY_3\\f_yY_2 + c_yY_3\end{bmatrix}\\ &= \frac{1}{(x'_2-c_y)\dfrac{d}{f_y}\sin\phi + d\cos\phi}\begin{bmatrix}(x'_1-c_x)d + c_x((x'_2-c_y)\dfrac{d}{f_y}\sin\phi + d\cos\phi)\\ f_y((x'_2-c_y)\dfrac{d}{f_y}\cos\phi - d\sin\phi) + c_y((x'_2-c_y)\dfrac{d}{f_y}\sin\phi + d\cos\phi)\end{bmatrix}\\ &= \frac{1}{(x'_2-c_y)\sin\phi + f_y\cos\phi}\begin{bmatrix}(x'_1-c_x)f_y + c_x((x'_2-c_y)\sin\phi + f_y\cos\phi)\\ f_y((x'_2-c_y)\cos\phi - f_y\sin\phi) + c_y((x'_2-c_y)\sin\phi + f_y\cos\phi)\end{bmatrix}\\ &= \frac{1}{(x'_2-c_y)\sin\phi + f_y\cos\phi}\begin{bmatrix}(x'_1-c_x)f_y + c_x((x'_2-c_y)\sin\phi + f_y\cos\phi)\\ f_y(x'_2\cos\phi - f_y\sin\phi) + c_y(x'_2-c_y)\sin\phi\end{bmatrix}\end{aligned} \tag{14.10}$$

Let Δ_ϕ be the 2D displacement due to the pitch ϕ.

$$\begin{aligned}\Delta_\phi &= \begin{bmatrix}y'_1\\y'_2\end{bmatrix} - \begin{bmatrix}x'_1\\x'_2\end{bmatrix}\\ &= \frac{1}{(x'_2-c_y)\sin\phi + f_y\cos\phi}\begin{bmatrix}f_y(x'_1-c_x) + (c_x-x'_1)((x'_2-c_y)\sin\phi + f_y\cos\phi)\\ -(x'^2_2 + f_y^2 + c_x^2)\sin(\phi)\end{bmatrix}\end{aligned} \tag{14.11}$$

It can be assumed that $f_x = f_y = f$ if the pixels are square which is generally true nowadays and $c_x = c_y = 0$ if the pixels are measured at the principal point of the image only. These two assumptions simplify the above equation as follows.

$$\begin{bmatrix}y'_1\\y'_2\end{bmatrix} = \frac{f}{x'_2\sin\phi + f\cos\phi}\begin{bmatrix}x'_1\\x'_2\cos\phi - f\sin\phi\end{bmatrix} \tag{14.12}$$

$$\begin{aligned}\Delta_\phi &= \begin{bmatrix}y'_1\\y'_2\end{bmatrix} - \begin{bmatrix}x'_1\\x'_2\end{bmatrix}\\ &= \frac{1}{x'_2\sin\phi + f\cos\phi}\begin{bmatrix}fx'_1(1-\cos\phi) - x'_1x'_2\sin\phi\\ -(x'^2_2 + f^2)\sin\phi\end{bmatrix}\end{aligned} \tag{14.13}$$

For the positive rotation ϕ, the displacement of the keypoint in the vertical direction of the image is always negative. It implies that the keypoint $[y'_1, y'_2]^T$ moves up in the image.

For the points at infinity i.e. $d \to \infty$, the 2D displacement Δ_ϕ does not change because it is independent of depth.

14.1.4 Rotation about $y-axis$ (Yaw)

In this case, the camera undergoes the rotation about the $y-axis$. Let θ be the yaw angle or the angle of rotation about the $y-axis$.

$$\begin{aligned}\begin{bmatrix}Y_1\\Y_2\\Y_3\end{bmatrix} &= \begin{bmatrix}\cos\theta & 0 & \sin\theta\\0 & 1 & 0\\-\sin\theta & 0 & \cos\theta\end{bmatrix}\begin{bmatrix}X_1\\X_2\\X_3\end{bmatrix}+\begin{bmatrix}0\\0\\0\end{bmatrix}\\ &= d\begin{bmatrix}\frac{x_1'-c_x}{f_x}\cos\theta+\sin\theta\\ \frac{x_2'-c_y}{f_y}\\ -\frac{(x_1'-c_x)}{f_x}\sin\theta+\cos\theta\end{bmatrix}\end{aligned} \tag{14.14}$$

$$\begin{aligned}\begin{bmatrix}y_1'\\y_2'\end{bmatrix} &= \frac{f_x}{-(x_1'-c_x)\sin\theta+f_x\cos\theta}\begin{bmatrix}(x_1'-c_x)\cos\theta+f_x\sin\theta+c_x\left(-\left(\frac{x_1'-c_x}{f_x}\right)\sin\theta+\cos\theta\right)\\ (x_2'-c_y)+c_y\left(-\left(\frac{x_1'-c_x}{f_x}\right)\sin\theta+\cos\theta\right)\end{bmatrix}\\ &= \frac{1}{-(x_1'-c_x)\sin\theta+f_x\cos\theta}\begin{bmatrix}f_xx_1'\cos\theta+(f_x^2+c_x^2)\sin\theta-c_xx_1'\sin\theta\\ f_xx_2'+f_xc_y(\cos\theta-1)-(x_1'-c_x)c_y\sin\theta\end{bmatrix}\end{aligned} \tag{14.15}$$

Let Δ_θ be the 2D displacement due to the yaw θ.

$$\Delta_\theta = \frac{1}{-(x_1'-c_x)\sin\theta+f_x\cos\theta}\begin{bmatrix}({x_1'}^2+f_x^2+c_x^2)\sin\theta-2x_1'c_x\sin\theta\\ (f_xx_2'-f_xc_y)(1-\cos\theta)+(x_1'-c_x)(x_2'-c_y)\sin\theta\end{bmatrix} \tag{14.16}$$

If we set $c_x=c_y=0$ and $f_x=f_y=f$, the above equations can be simplified and results into

$$\begin{bmatrix}y_1'\\y_2'\end{bmatrix} = \frac{f}{-x_1'\sin\theta+f\cos\theta}\begin{bmatrix}x_1'\cos\theta+f\sin\theta\\ x_2'\end{bmatrix} \tag{14.17}$$

$$\Delta_\theta = \frac{1}{-x_1'\sin\theta+f\cos\theta}\begin{bmatrix}({x_1'}^2+f^2)\sin\theta\\ fx_2'(1-\cos\theta)+x_1'x_2'\sin\theta\end{bmatrix} \tag{14.18}$$

For the positive rotation θ, the displacement of the keypoint in the horizontal direction of the image is always positive. It implies that the keypoint $[y_1', y_2']^T$ moves right in the image.

For the points in infinity i.e. $d\to\infty$, the 2d motion ($=\Delta_\theta$) does not change because it is independent of depth.

14.1.5 Rotation about $z-axis$ (Roll)

In this case, the camera undergoes rotation about $z-axis$. Let ψ be the roll angle or the angle of rotation about the $z-axis$.

$$\begin{aligned}\begin{bmatrix}Y_1\\Y_2\\Y_3\end{bmatrix} &= \begin{bmatrix}\cos\psi & -\sin\psi & 0\\ \sin\psi & \cos\psi & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}X_1\\X_2\\X_3\end{bmatrix} + \begin{bmatrix}0\\0\\0\end{bmatrix}\\ &= d\begin{bmatrix}\left(\frac{x_1'-c_x}{f_x}\right)\cos\psi - \left(\frac{x_2'-c_y}{f_y}\right)\sin\psi\\ \left(\frac{x_1'-c_x}{f_x}\right)\sin\psi + \left(\frac{x_2'-c_y}{f_y}\right)\cos\psi\\ 1\end{bmatrix}\end{aligned} \tag{14.19}$$

$$\begin{aligned}\begin{bmatrix}y_1'\\y_2'\end{bmatrix} &= \begin{bmatrix}(x_1'-c_x)\cos\psi - \frac{f_x(x_2'-c_y)\sin\psi}{f_y} + c_x\\ \frac{f_y(x_1'-c_x)\sin\psi}{f_x} + (x_2'-c_y)\cos\psi + c_y\end{bmatrix}\\ &= \begin{bmatrix}\frac{f_y(x_1'-c_x)\cos\psi - f_x(x_2'-c_y)\sin\psi + f_yc_x}{f_y}\\ \frac{f_y(x_1'-c_x)\sin\psi + f_x(x_2'-c_y)\cos\psi + f_xc_y}{f_x}\end{bmatrix}\end{aligned} \tag{14.20}$$

Let Δ_ψ be the 2D displacement due to the roll ψ.

$$\Delta_\psi = \begin{bmatrix}\frac{f_y(x_1'-c_x)\cos\psi - f_x(x_2'-c_y)\sin\psi + f_yc_x - f_yx_1'}{f_y}\\ \frac{f_y(x_1'-c_x)\sin\psi + f_y(x_2'-c_y)\cos\psi + f_xc_y - f_xx_2'}{f_x}\end{bmatrix} \tag{14.21}$$

If we set $c_x = c_y = 0$ and $f_x = f_y = f$, the above equations can be simplified and results into

$$\begin{bmatrix}y_1'\\y_2'\end{bmatrix} = \begin{bmatrix}x_1'\cos\psi - x_2'\sin\psi\\ x_1'\sin\psi + x_2'\cos\psi\end{bmatrix} \tag{14.22}$$

$$\Delta_\psi = \begin{bmatrix}x_1'(\cos\psi - 1) - x_2'\sin\psi\\ x_1'\sin\psi + x_2'(\cos\psi - 1)\end{bmatrix} \tag{14.23}$$

Let $\mathbf{R}_\psi$ be the 2×2 rotation matrix and $\mathbf{I}$ be the identity matrix then the eq. 14.22 and 14.23 can be re-written in the compact form.

$$\vec{y}' = \mathbf{R}_\psi \cdot \vec{x}' \tag{14.24}$$

$$\Delta_\psi = (\mathbf{R}_\psi - \mathbf{I})\vec{x}' \tag{14.25}$$

For the points in infinity i.e. $d \to \infty$, the 2d motion ($=\Delta_\psi$) does not change because it is independent of depth.

14.1.6 Translation along $x-axis$

In this case, the camera undergoes the translation along $x-axis$. Let t_1 be the translation along the $x-axis$.

$$\begin{aligned}\begin{bmatrix}Y_1\\Y_2\\Y_3\end{bmatrix} &= \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}X_1\\X_2\\X_3\end{bmatrix} + \begin{bmatrix}t_1\\0\\0\end{bmatrix}\\ &= d\begin{bmatrix}\frac{x_1'-c_x}{f_x}+\frac{t_1}{d}\\ \frac{x_2'-c_y}{f_y}\\1\end{bmatrix}\end{aligned} \tag{14.26}$$

$$\begin{bmatrix}y_1'\\y_2'\end{bmatrix} = \begin{bmatrix}x_1'+\frac{t_1 f_x}{d}\\ x_2'\end{bmatrix} \tag{14.27}$$

Let Δ_{t_1} be the 2D displacement due to the translation t_1.

$$\Delta_{t_1} = \begin{bmatrix}\frac{f_x t_1}{d}\\0\end{bmatrix} \tag{14.28}$$

If we set $c_x = c_y = 0$ and $f_x = f_y = f$, the above equations can be simplified and results into

$$\begin{bmatrix}y_1'\\y_2'\end{bmatrix} = \begin{bmatrix}x_1'+\frac{t_1 f}{d}\\ x_2'\end{bmatrix} \tag{14.29}$$

$$\Delta_{t_1} = \begin{bmatrix}\frac{f t_1}{d}\\0\end{bmatrix} \tag{14.30}$$

The displacement is positive for positive translation and vice-versa. For the points in infinity i.e. $d \to \infty$, the 2d motion vanishes as displacement approaches zero.

$$\lim_{d\to\infty} \Delta_{t_1} = \begin{bmatrix}0\\0\end{bmatrix} \tag{14.31}$$

14.1.7 Translation along $y-axis$

In this case, the camera undergoes the translation along the $y-axis$. Let t_2 be the translation along the $y-axis$.

$$\begin{aligned}\begin{bmatrix}Y_1\\Y_2\\Y_3\end{bmatrix} &= \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}X_1\\X_2\\X_3\end{bmatrix} + \begin{bmatrix}0\\t_2\\0\end{bmatrix}\\ &= d\begin{bmatrix}\frac{x_1'-c_x}{f_x}\\ \frac{x_2'-c_y}{f_y}+\frac{t_2}{d}\\1\end{bmatrix}\end{aligned} \tag{14.32}$$

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 + \dfrac{f_y t_2}{d} \end{bmatrix} \tag{14.33}$$

Let Δ_{t_2} be the 2D displacement due to the translation t_2.

$$\Delta_{t_2} = \begin{bmatrix} 0 \\ \dfrac{f_y t_2}{d} \end{bmatrix} \tag{14.34}$$

If we set $c_x = c_y = 0$ and $f_x = f_y = f$, the above equations can be simplified and results into

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 + \dfrac{f t_2}{d} \end{bmatrix} \tag{14.35}$$

$$\Delta_{t_2} = \begin{bmatrix} 0 \\ \dfrac{f t_2}{d} \end{bmatrix} \tag{14.36}$$

The displacement is positive for positive translation and vice-versa. For the points in infinity i.e. $d \to \infty$, the 2d motion vanishes as displacement approaches zero.

$$\lim_{d \to \infty} \Delta_{t_2} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{14.37}$$

14.1.8 Translation along z-axis

In this case, the camera undergoes the translation along the $z - axis$. Let t_3 be the translation along the $z - axis$.

$$\begin{aligned} \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{bmatrix} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ t_3 \end{bmatrix} \\ &= d \begin{bmatrix} \dfrac{x'_1 - c_x}{f_x} \\ \dfrac{x'_2 - c_y}{f_y} \\ 1 + \dfrac{t_3}{d} \end{bmatrix} \end{aligned} \tag{14.38}$$

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \frac{d}{d + t_3} \begin{bmatrix} x'_1 + \dfrac{c_x t_3}{d} \\ x'_2 + \dfrac{c_y t_3}{d} \end{bmatrix} \tag{14.39}$$

Let Δ_{t_3} be the 2D displacement due to the translation t_3.

$$\Delta_{t_3} = \frac{1}{d + t_3} \begin{bmatrix} t_3(c_x - x'_1) \\ t_3(c_y - x'_2) \end{bmatrix} \tag{14.40}$$

If we set $c_x = c_y = 0$ and $f_x = f_y = f$, the above equations can be simplified and results into

$$\begin{bmatrix} y'_1 \\ y'_2 \end{bmatrix} = \frac{d}{d + t_3} \begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} \tag{14.41}$$

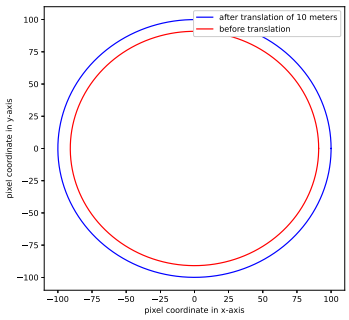


Figure 14.3: Expansion field from translation along $z - axis$

$$\Delta_{t_3} = \frac{-1}{d + t_3} \begin{bmatrix} t_3 x_1' \\ t_3 x_2' \end{bmatrix} \tag{14.42}$$

For the positive values of $[x_1', x_2']^T$, the displacement is always negative for positive displacement and vice-versa. For the points in infinity i.e. $d \to \infty$, the 2d motion vanishes as displacement approaches zero.

$$\lim_{d \to \infty} \Delta_{t_3} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{14.43}$$

The motion field generated from the translation along the $z - axis$ can be visualized as an expansion field. If the camera sees a circle and then translates forward (i.e. along the $z - axis$), then the circle starts expanding in the image (see Fig. 14.3).

14.2 What can be considered a 'far away region'?

As we already observed from the previous sections that if the point is at infinity, the contribution of the translation motion to the 2D motion is zero, however, it is not practically possible to identify the points at infinity. Therefore, it is very crucial to set some thresholds on the depth such that if the point is farther than this threshold, then we can assume that the 2D displacement happened due to rotation only.

One of the approaches can be limiting the maximum displacement caused by the translation. Let t be the translation along any axis, Δ_{max} be the maximum allowed 2D displacement of the pixel due to translation only, and d_{th} be the corresponding depth threshold, then the depth threshold for translation motion along each axis can be computed using the equations 14.30, 14.36, and 14.42.

14.2.1 Estimating depth threshold from translation along x-axis

In equation 14.30, we can see that the translation causes the displacement in the x_1' only. From this, we can find the corresponding d_{th} as follows.

$$d_{th} = \frac{ft}{\Delta_{max}} \tag{14.44}$$

Table 14.1: Parameters of the camera to study depth thresholds (approximate values of the ZED 1 camera)

Parameter name	code name	value
FPS	fps	15
Image resolution	res	2208×1242
Focal length	f	1400 pixels

14.2.2 Estimating depth threshold from translation along y-axis

In equation 14.36, we can see that the translation causes the displacement in the x_2' only. From this, we can find the corresponding d_{th} as follows.

$$d_{th} = \frac{ft}{\Delta_{max}} \tag{14.45}$$

14.2.3 Estimating depth threshold from translation along z-axis

It is not very straightforward to compute the d_{th} from translation and displacement only as it is also a function of the pixel coordinates (see eq. 14.42). From the eq. 14.42, it is clear that the displacement is directly proportional to the value of the pixel itself. If I want to limit the maximum value of displacement in the image, then the focus should be on the maximum value of the pixel coordinate.

$$x_{max} = \begin{cases} x_1' & \text{for} \quad |x_1'| > |x_2'| \\ x_2' & \text{for} \quad otherwise \end{cases} \tag{14.46}$$

Once, we have the maximum value, we can calculate the d_{th} using the following equation

$$d_{th} = -\frac{t(x_{max} - sign(x_{max})\Delta_{max})}{-sign(x_{max})\Delta_{max}} \tag{14.47}$$

where,

$$sign(x) = \begin{cases} 1 & \text{for} \quad x \geq 0 \\ -1 & \text{for} \quad otherwise \end{cases} \tag{14.48}$$

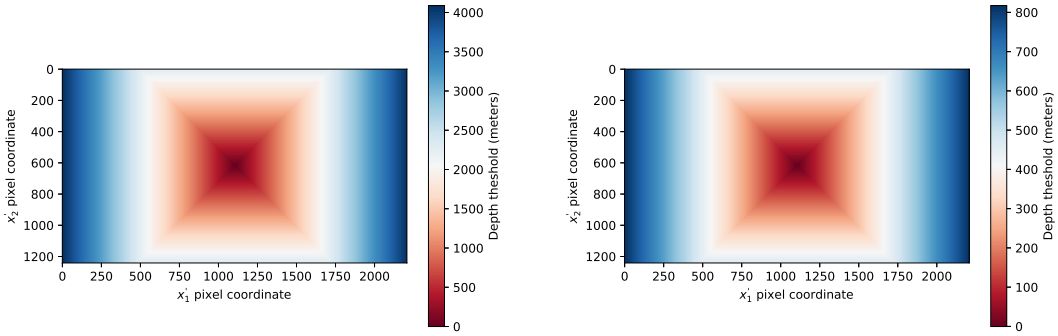
The sign function is important to keep the depth threshold d_{th} positive. The maximum translation displacement Δ_{max} always has the opposite sign of pixel coordinate for positive translation (see eq. 14.42).

14.2.4 Depth threshold estimation for vehicles

In this section, I will consider two values for the maximum displacement Δ_{max} and estimate the depth threshold d_{th} for the cars and the ships.

In the general case, the translation motion of cars and ships is restricted to the $z - axis$ only and therefore, I considered the translation along the $z - axis$ to estimate the depth threshold d_{th}. Let s be the speed of the ego-vehicle in Km/h and fps be the FPS of the camera, and t be the translation of the ego-vehicle between two consecutive frames then I can compute the translation t for different speeds of the ego-vehicle for a camera with a FPS fps.

$$t = \frac{s}{fps} \tag{14.49}$$



(a) 0.1 pixels of maximum displacement (b) 0.5 pixels of maximum displacement

Figure 14.4: Depth threshold field for the ego-vehicle moving at 20 km/h

As depth threshold d_{th} is a function of the pixel coordinates, the threshold value is not constant for the whole image and varies from point to point. Figure 14.4 shows the depth threshold field for the whole image with a resolution *res*. It can be noticed that the shape of the field remains the same but the range of the depth threshold d_{th} changes with the change in the maximum allowed displacement Δ_{max}. When the Δ_{max} increases, the maximum value of the depth threshold d_{th} in the field decreases. In case $\Delta_{max} = 0.1$ (see Fig.. 14.4a), the maximum approximate value for the depth threshold d_{th} is 4000 meters but for $\Delta_{max} = 0.5$ (see Fig.. 14.4b), the maximum approximate value for the depth threshold d_{th} is 800 meters.

Similar results can be generated for cars moving at 50 km/h, 100 km/h, and 150 km/h or for ferries moving with speeds of 3, 6, and 10 knots. Figure 14.5 shows the depth threshold field for different speeds and the maximum displacement Δ_{max} with a common scale for the ferry. From the results, it can be inferred that the depth threshold increases with the increase of the speed s of the car, the maximum displacement Δ_{max}, and decreases when the pixel is towards the center of the image.

14.3 Conclusions on the dependencies of image plane displacements on 3D motion and 3D depth

From the displacement equations for rotations, it can be observed that the displacement is not related to the depth of the point at all in the case of pure rotations. This also happens when we do stereo registration in which there is a virtual rotation of both cameras until their viewing directions are exactly parallel. Also, the displacement becomes independent of translation motion when depth d moves towards infinity. In other words, when depth d goes to infinity, the displacement becomes zero for translation along the $x - axis$, $y - axis$, and $z - axis$.

14.4 Geometrical interpretation of the yaw and pitch

[2] used the geometrical interpretation to estimate the yaw and pitch angles which is explained in this section. Let $\vec{x}$ be the world point, p_1 and p_2 be the projections of the world point $\vec{x}$ on

two consecutive images, α be the rotational motion of the camera between two images (see Fig. 14.6a), then the angle α can be computed geometrically.

$$\alpha = \text{atan2}(-p_2, f) - \text{atan2}(-p_1, f) \tag{14.50}$$

The points p_1 and p_2 can be converted into pixel coordinates p_1' and p_2' respectively using the principal point c from the intrinsics of the camera.

$$p = p' - c \tag{14.51}$$

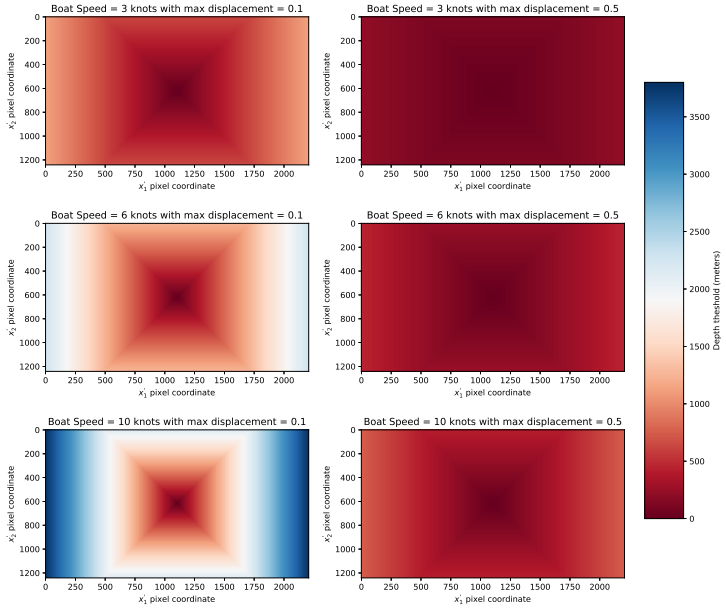
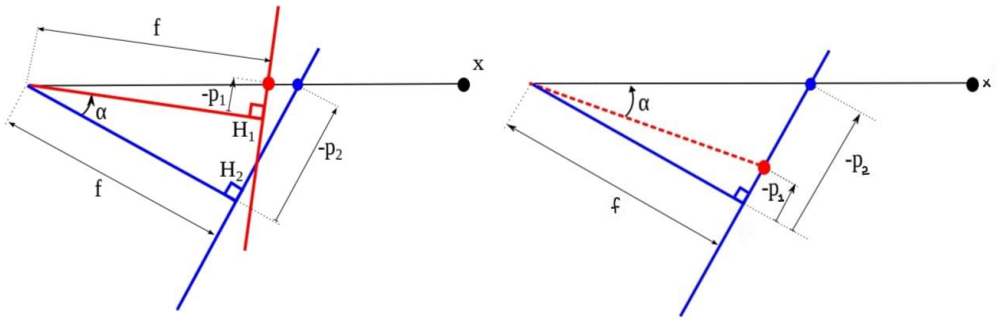


Figure 14.5: Depth threshold field for the boat/ferry moving at different speeds. The value of the Δ_{max} is 0.1 pixels and 0.5 pixels for the left and right figures respectively.



(a) Two unaligned images viewing the same point x [2]

(b) Aligned images (Corrected version from [2])

Figure 14.6: Estimation of 2D rotation from pixel displacement

Eq. 14.50 is generalized and can be used to compute yaw θ and pitch ψ using displacement of matched pixels along horizontal and vertical axis.

$$\theta = \text{atan2}(c_x - p'_{1x}, f_x) - \text{atan2}(c_x - p'_{2x}, f_x) \tag{14.52}$$

$$\psi = \text{atan2}(c_y - p'_{2y}, f_y) - \text{atan2}(c_y - p'_{1y}, f_y) \tag{14.53}$$

14.5 Rotation estimation from 2D motion

In the previous sections, we saw how the 2D displacement can be computed if the motion is known. In this section, I reversed the problem and derived expressions to estimate the rotation from the 2D displacement. The expressions derived in section 14.1 are used to estimate the rotation motion.

14.5.1 Estimation of rotation about $x - axis$ (Pitch)

In eq. 14.12, we found the expression to map the pixel coordinates from one image to another after going through some rotation around the $x - axis$. I can utilize only the y'_2 component from the equation to find the desired angle of rotation.

$$\begin{aligned} y'_2 &= \frac{f_y(x'_2 \cos\phi - f_y \sin\phi)}{x'_2 \sin\phi + f_y \cos\phi} \\ y'_2(x'_2 \tan\phi + f_y) &= f_y(x'_2 - f_y \tan\phi) \\ \phi &= \text{atan2}\left(\frac{x'_2 f_y - y'_2 f_y}{y'_2 x'_2 + f_y^2}\right) \end{aligned} \tag{14.54}$$

The results obtained in the above equation are equivalent to the one given in [2].

$$\begin{aligned} \phi &= \text{atan2}\left(\frac{-y'_2}{f_y}\right) - \text{atan2}\left(\frac{-x'_2}{f_y}\right) \\ \phi &= \text{atan2}\left(\tan\left(\text{atan2}\left(\frac{-y'_2}{f_y}\right) - \text{atan2}\left(\frac{-x'_2}{f_y}\right)\right)\right) \\ \phi &= \text{atan2}\left(\frac{\frac{-y'_2}{f_y} - \frac{-x'_2}{f_y}}{1 + \frac{-y'_2}{f_y}\frac{-x'_2}{f_y}}\right) \\ \phi &= \text{atan2}\left(\frac{x'_2 f_y - y'_2 f_y}{y'_2 x'_2 + f_y^2}\right) \end{aligned} \tag{14.55}$$

It should be noted that I assumed $c_x = c_y = 0$ which implies that the points $[x'_1, x'_2]$ and $[y'_1, y'_2]$ are measured w.r.t the principle point of the camera. If they are given in pixel coordinates i.e. measured with respect to the top-left of the image, then the equation to estimate the pitch angleis the following.

$$\phi = \text{atan2}\left(\frac{-(y'_2 - c_y)}{f_y}\right) - \text{atan2}\left(\frac{-(x'_2 - c_y)}{f_y}\right) \tag{14.56}$$

14.5.2 Estimation of rotation about $y-axis$ (Yaw)

Similar to the previous section, the eq. 14.17 can be exploited to get the angle of rotation about the $y-axis$. For this case, I utilized only the y_1' component from the equation to find the desired angle of rotation.

$$\begin{aligned} y_1' &= \frac{f_x(x_1' \cos\theta + f_x \sin\theta)}{-x_1' \sin\theta + f_x \cos\theta} \\ y_1'(-x_1' \tan\theta + f_x) &= f_x(x_1' + f_x \tan\theta) \\ \theta &= \text{atan2}\left(\frac{y_1' f_x - x_1' f_x}{y_1' x_1' + f_x^2}\right) \end{aligned} \tag{14.57}$$

Again, the equation given in [2] to compute the yaw can be expanded to compare it with my derivation.

$$\begin{aligned} \theta &= \text{atan2}\left(\frac{-y_1'}{f_x}\right) - \text{atan2}\left(\frac{-x_1'}{f_x}\right) \\ &= \text{atan2}\left(\tan\left(\text{atan2}\left(\frac{-y_1'}{f_x}\right) - \text{atan2}\left(\frac{-x_1'}{f_x}\right)\right)\right) \\ &= \text{atan2}\left(\frac{\frac{-y_1'}{f_x} - \frac{-x_1'}{f_x}}{1 + \frac{-y_1'}{f_x}\frac{-x_1'}{f_x}}\right) \\ &= \text{atan2}\left(\frac{x_1' f_x - y_1' f_x}{y_1' x_1' + f_x^2}\right) \end{aligned} \tag{14.58}$$

If we notice carefully, the rotation angles I obtained from the derivation and from the original paper are equal in magnitude but opposite in direction. This is a technical error in the paper. The corrected equation to compute the yaw motion should be.

$$\theta = \text{atan2}\left(\frac{-x_1'}{f_x}\right) - \text{atan2}\left(\frac{-y_1'}{f_x}\right) \tag{14.59}$$

If $c_x \neq 0$ then the revised expression is

$$\theta = \text{atan2}\left(\frac{-(x_1' - c_x)}{f_x}\right) - \text{atan2}\left(\frac{-(y_1' - c_x)}{f_x}\right) \tag{14.60}$$

14.5.3 Estimation of rotation about $z-axis$ (Roll)

Similar to the previous section, the eq. 14.22 can be exploited to get the angle of rotation about the $z-axis$. For this case, I utilized both y_1' and y_2' components from the equation to find the desired angle of rotation

$$\begin{aligned} \frac{y_1'}{y_2'} &= \frac{x_1' \cos\psi - x_2' \sin\psi}{x_1' \sin\psi + x_2' \cos\psi} \\ \psi &= \text{atan2}\left(\frac{y_2' x_1' - y_1' x_2'}{y_1' x_1' + y_2' x_2'}\right) \end{aligned} \tag{14.61}$$

In [2], the authors solved the estimation problem by considering it a *Procrustes problem*. Procrustes problem or Orthogonal Procrustes problem is the least-squares problem of transforming a given matrix $\mathbf{A}$ into a given matrix $\mathbf{B}$ by an orthogonal transformation matrix $\mathbf{T}$ such that the sums of the squares of the residual matrix $\mathbf{E} = \mathbf{AT} - \mathbf{B}$ ([41]).

The solution to solve the Procrustes problem using the Singular Value Decomposition (SVD) is given in [13]. If I assume $\vec{c}_i$ and $\vec{c}'_i$ represent the pixel coordinates of the correspondences from the frame I_{t-1} and I_t respectively and $\vec{d}_i$ is the 2D displacement vector then, I can define two corresponding point sets $\vec{c}_i \in \mathcal{C}$ and $\vec{c}'_i \in \mathcal{C}'$, such that their elements are related by

$$\vec{c}'_i = \vec{c}_i + \vec{d}_i \tag{14.62}$$

To calculate the rotation, I translated the point sets $\mathcal{C}$ and $\mathcal{C}'$ to the origin of the coordinate system by subtracting the mean of the point cloud.

$$\begin{aligned} \bar{\vec{c}} &= \frac{1}{N}\sum_{i=1}^{N} \vec{c}_i \quad \longrightarrow \quad \hat{\vec{c}}_i = \vec{c}_i - \bar{\vec{c}} \\ \bar{\vec{c}}' &= \frac{1}{N}\sum_{i=1}^{N} \vec{c}'_i \quad \longrightarrow \quad \hat{\vec{c}}'_i = \vec{c}'_i - \bar{\vec{c}}' \end{aligned} \tag{14.63}$$

Let $\mathbf{R}$ be a 2×2 rotation matrix defined for the roll angle ψ.

$$\mathbf{R} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \tag{14.64}$$

Let $\mathbf{A} = \left[\hat{\vec{c}}_1, \hat{\vec{c}}_2, ..., \hat{\vec{c}}_N,\right]$ and $\mathbf{B} = \left[\hat{\vec{c}}'_1, \hat{\vec{c}}'_2, ..., \hat{\vec{c}}'_N,\right]$ be the $2 \times N$ matrices defined by stacking elements (row-wise) of $\mathcal{C}$ and $\mathcal{C}'$ respectively then the objective or the loss functions defined in [13] and [2] to estimate the roll angle can be written as eq. 14.65 and eq. 14.66 respectively.

$$\vec{e}^2 = \frac{1}{N}\sum_{i=1}^{N} ||\hat{\vec{c}}'_i - \mathbf{R}\hat{\vec{c}}_i||^2 \tag{14.65}$$

$$min||\mathbf{AR} - \mathbf{B}||_F \tag{14.66}$$

The loss function defined in 14.66 and used by [2] is similar to the one used in [41], however, I believe that the authors of [2] defined it wrongly because the rotation matrix should have been either pre-multiplied to the matrix $\mathbf{A}$ or the inverse of the rotation matrix should have been used.

Explanation: If I assume, the post-multiplication and pre-multiplication of the rotation matrix give the same results then $\mathbf{B} = \mathbf{RA}$ should be equivalent to $\mathbf{B}^T = \mathbf{A}^T\mathbf{R}$ (the transpose changes the matrix from column-major to row-major). then the following should be true as well.

$$(\mathbf{B}^T)^T = (\mathbf{A}^T\mathbf{R})^T \rightarrow \mathbf{B} = \mathbf{R}^T\mathbf{A} \tag{14.67}$$

It implies that the $\mathbf{R} = \mathbf{R}^T$ is possible only if the rotation matrix is the identity matrix.

Example: Let us assume if $\mathbf{A}$ contains only one point $[x_1, x_2]$ then $\mathbf{A}$ can be either $[x_1, x_2]$ or $[x_1, x_2]^T$. If $\mathbf{A}$ is pre-multiplied with the rotation matrix $\mathbf{R}$ then we get the following result.

$$\mathbf{RA} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1\cos\psi - x_2\sin\psi \\ x_1\sin\psi + x_2\cos\psi \end{bmatrix} \tag{14.68}$$

however, if $\mathbf{R}$ is pose multiplied with $\mathbf{A}$, we obtain different results

$$\mathbf{AR} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} = \begin{bmatrix} x_1\cos\psi + x_2\sin\psi & -x_1\sin\psi + x_2\cos\psi \end{bmatrix} \tag{14.69}$$

From the above two results, it is clear that post-multiplying rotates the point in the opposite direction. Therefore, the definition of the rotation matrix $\mathbf{R}$ in [2] is either wrong or the object function (see eq. 14.66) is wrong.

I used eq. 14.65 as the loss function that has to be minimized. If I expand the objective function, I get the following.

$$\vec{e}^2 = \sum_{i=1}^{N} (\hat{\vec{c}}_i'^T \hat{\vec{c}}_i' + \hat{\vec{c}}_i^T \hat{\vec{c}}_i - 2\hat{\vec{c}}_i'^T \mathbf{R} \hat{\vec{c}}_i) \tag{14.70}$$

This equation is minimized when the last term $\hat{\vec{c}}_i'^T \mathbf{R} \hat{\vec{c}}_i$ is maximized which is equivalent to maximizing $Trace(\mathbf{RH})$, where $\mathbf{H}$ is the correlation matrix.

$$\begin{aligned} \mathbf{H} &= \sum_{i=1}^{N} \hat{\vec{c}}_i \hat{\vec{c}}_i'^T \\ &= \mathbf{A}\mathbf{B}^T \end{aligned} \tag{14.71}$$

I computed the SVD of correlation matrix $\mathbf{H}$ and get the orthogonal vectors as follows.

$$\mathbf{U}, \Lambda, \mathbf{V}^T = SVD(\mathbf{H}) \tag{14.72}$$

Finally, the rotation matrix $\mathbf{R}$ can be computed.

$$\mathbf{R} = \mathbf{V}\mathbf{U}^T \tag{14.73}$$

Analysis of the roll estimation

It is necessary to perform a few tests to validate the working and accuracy of the solution to the Procrustes algorithm to compute the roll angle. In this test, I generated a set of N 2D points from a Gaussian distribution. The distribution has the mean μ and the standard deviation σ. Once I have the 2D points, the matrix $\mathbf{A}$ can be initialized by stacking the points row-wise. A random roll motion ψ was applied to $\mathbf{A}$ and corresponding matched points (let us call it $\mathbf{B}$) are generated.

$$\mathbf{A} = Gauss(\mu, \sigma, size = (2, N)) \tag{14.74}$$

$$\mathbf{B} = \begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \mathbf{A} \tag{14.75}$$

$\mathbf{A}$, $\mathbf{B}$, and ψ represent the ideal situation where the points are perfectly matched and the location is exactly known. To test the algorithm, I can assume that the points are perfectly matched but with an approximate location so I added a random noise with mean μ_n and standard deviation σ_n to $\mathbf{A}$ and $\mathbf{B}$.

$$\begin{aligned} \mathbf{A}' &= \mathbf{A} + Gauss(\mu_n, \sigma_n, size = (2, N)) \\ \mathbf{B}' &= \mathbf{B} + Gauss(\mu_n, \sigma_n, size = (2, N)) \end{aligned} \tag{14.76}$$

Once, the noisy data is generated, I standardized the sets by subtracting the mean (ref eq. 14.63) and computed the correlation matrix $\mathbf{H}$ (ref eq. 14.71). The rotation matrix $\mathbf{R}$ was estimated by applying eq 14.72, and eq. 14.73. Finally, the roll is estimated from the rotation matrix using the following equation.

$$\tilde{\phi} = \text{atan2}(\sin\psi, \cos\psi) = \text{atan2}(\mathbf{R}[1,0], \mathbf{R}[0,0]) \tag{14.77}$$

Table 14.2: Parameters used in analysis of the roll

Parameter	Value
μ	0
σ	100
N	4
μ_n	0
σ_n	1

Once the test pipeline is set, The influence of different parameters on the accuracy of roll estimation can be analyzed. I evaluated four parameters against the absolute roll error. These parameters are the standard deviation of point set distribution σ, number of matched points N, mean of noise μ_n, and standard deviation of noise σ_n. The effect of the parameters on the absolute roll error has been plotted in Fig. 14.7. From figure 14.7, the following statements can be made.

- The greater the standard deviation in the point set, the less the error will be. The error doesn't decrease much after $\sigma = 9$ and decreases very slowly after $\sigma = 20$.
- The error is very large if the number of correspondences is below 4. A similar observation was also given in [2].
- The mean of noise has no direct influence on the error or shows no pattern.
- The error is directly proportional to the standard deviation of noise added to the data. Hence, the accuracy in the location of points is very crucial to the estimation of error.

14.6 Implementation of the rotation estimator

The estimation of roll, pitch, and yaw angle from the far-away corresponding points has been summarised in this section. Let $\vec{c}_i$ and $\vec{c}'_i$ be the pixel coordinates of the corresponding points from the frame I_{t-1} and I_t respectively then, I can define two corresponding point sets $\mathcal{C}$ and $\mathcal{C}'$ such that $\vec{c}_i \in \mathcal{C}$ and $\vec{c}'_i \in \mathcal{C}'$.

It should be noted that unlike the roll estimation which uses all the corresponding points at once by treating it as a Procrustes problem, I will get one pitch ϕ and yaw θ angle for every pair of correspondence $(\vec{c}_i, \vec{c}'_i)$. Let $[x'_{1i}, x'_{2i}]^T$ and $[y'_{1i}, y'_{2i}]^T$ be the pixel coordinates defined by $\vec{c}_i$ and $\vec{c}'_i$ respectively, ϕ_i and θ_i be the pitch and yaw angle estimated for the correspondence pair $(\vec{c}_i, \vec{c}'_i)$ respectively, and N be the total number of correspondences, then the pitch ϕ_i and yaw θ_i can be computed.

$$\phi_i = \text{atan2}\left(\frac{-(y'_{2i} - c_y)}{f_y}\right) - \text{atan2}\left(\frac{-(x'_{2i} - c_y)}{f_y}\right) \quad \forall\ i = 1...N \tag{14.78}$$

$$\theta_i = \text{atan2}\left(\frac{-(x'_{1i} - c_x)}{f_x}\right) - \text{atan2}\left(\frac{-(y'_{1i} - c_x)}{f_x}\right) \quad \forall\ i = 1...N \tag{14.79}$$

One potential solution to obtain the final pitch and yaw is to take the mean of the arrays. This gives equal weight to each estimated angle. Instead of taking the mean, I computed the weighted average where I weighted them according to their *correspondence scores*. The correspondence

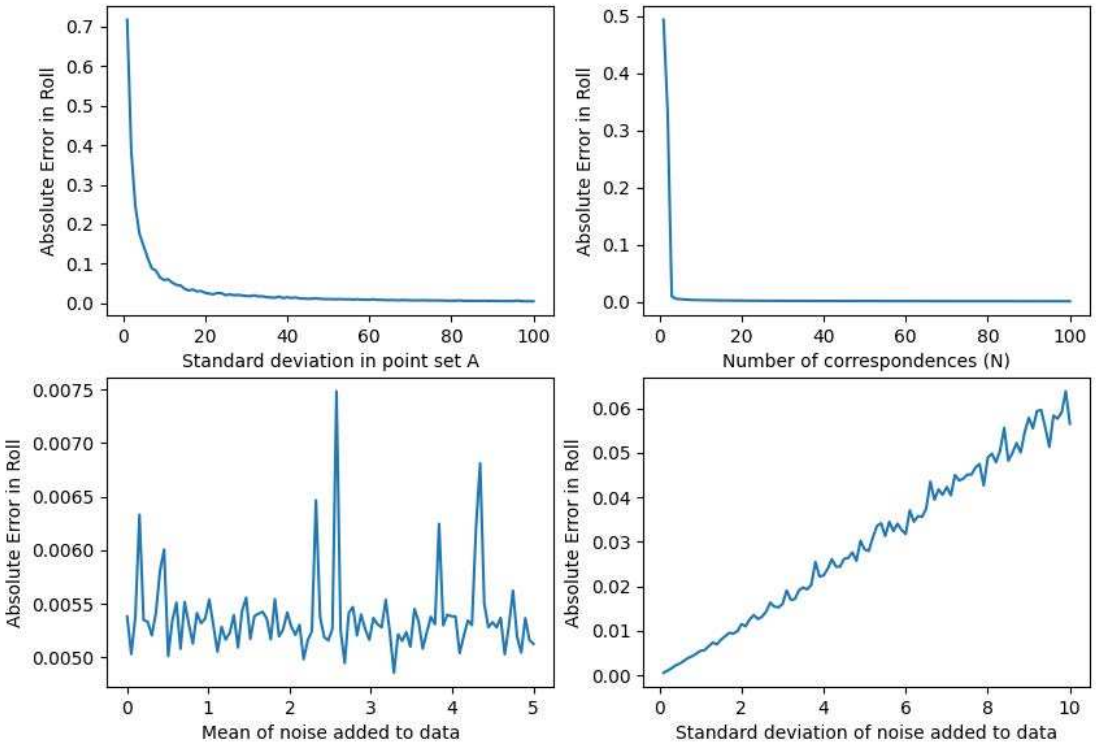


Figure 14.7: Analysis of Roll Estimation Algorithm

score is a measure of the quality of the correspondences. If the correspondence has more residual error, then its score will be proportionally less, and therefore, the angle measured from this correspondence pair will be given less weightage. Let $\bar{\phi}$ be the final estimated pitch angle and $\bar{\theta}$ be the final estimated yaw angle, and $\alpha = \alpha_i \ \forall i = 1...N$ be the set of the correspondences' scores then the final pitch $\bar{\phi}$ and yaw angle $\bar{\theta}$ can be computed by taking the weighted average of the array of the angles.

$$\bar{\phi} = \frac{\sum_{i=1}^{N} \alpha_i \phi_i}{\sum_{i=1}^{N} \alpha_i} \tag{14.80}$$

$$\bar{\theta} = \frac{\sum_{i=1}^{N} \alpha_i \theta_i}{\sum_{i=1}^{N} \alpha_i} \tag{14.81}$$

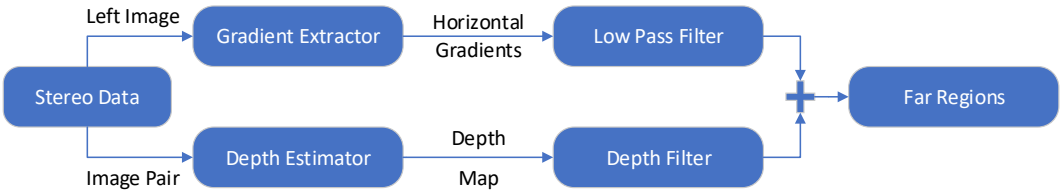


Figure 14.8: Flow chart of an algorithm for visual inspection of far-away regions

Algorithm 15 Estimation of roll, pitch, and yaw

procedure $get\_rpy(\mathcal{C}, \mathcal{C}', \alpha)$
  \\ Pitch Estimation
  $[\phi] = \textbf{compute\_pitch}(\mathcal{C}, \mathcal{C}', c_y, f_y)$ ▷ see eq. 14.78
  $\bar{\phi} = \textbf{weighted\_average}([\phi], \alpha)$ ▷ see eq. 14.80
  \\ Yaw Estimation
  $[\theta] = \textbf{compute\_yaw}(\mathcal{C}, \mathcal{C}', c_x, f_x)$ ▷ see eq. 14.79
  $\bar{\theta} = \textbf{weighted\_average}([\theta], \alpha)$ ▷ see eq. 14.81
  \\ Roll Estimation
  $\mathbf{A} = \textbf{origin\_shift}(\mathcal{C})$ ▷ see eq. 14.63
  $\mathbf{B} = \textbf{origin\_shift}(\mathcal{C}')$ ▷ see eq. 14.63
  $\mathbf{H} = \mathbf{A}\mathbf{B}^T$ ▷ see eq. 14.71
  $\mathbf{U}, \Lambda, \mathbf{V}^T = \textbf{SVD}(\mathbf{H})$ ▷ see eq. 14.72
  $\mathbf{R} = \mathbf{V}\mathbf{U}^T$ ▷ see eq. 14.73
  $\phi = \text{atan2}(\mathbf{R}[1,0], \mathbf{R}[0,0])$ ▷ see eq. 14.77
  return $\phi, \bar{\phi}, \bar{\theta}$

14.7 Identification and visualization of far-away regions

The accuracy of the estimated rotational angles highly depends only on the fact that the region or image points under consideration are far enough. Unlike the monocular case, I can utilize the depth maps from the stereo camera to filter out the far-away regions. The disparity map or the depth map may have holes due to occlusions or wrong estimates due to poor illumination, low resolution, etc. Therefore, it becomes a primary task to verify whether the disparity map can be trusted or not.

The disparity map can be obtained using the *Semi Global Block Matching (SGBM)* method or advanced methods such as Neural Network (NN). The accuracy of the former approach suffers from the textureless surfaces and the latter tries to assign depth even to objectively non-measurable regions such as the sky. In this project, I used a ZED camera that computes the disparity map using its *Ultra* mode (see section D.2, p.184 for the definition), therefore, I tried

to find the far-away areas using ZED disparity maps and verified visually if these far regions in the disparity map are actually far or not.

In our actual vision pipeline, I am focusing on the regions that have some texture and avoiding the textureless surfaces. In these regions, the reliability of the disparity map also increases. Therefore, I want to verify the disparity map for far-away regions with some texture. Keeping the depth and texture as constraints to find the far regions in the image, I decided to apply these constraints independently to the input data and merge the results to get reliable far-away regions for visual inspection. The algorithm is shown in Fig. 14.8.

To find the texture, I first computed the horizontal gradients of the left image. The motivation to estimate only horizontal gradients is that in the case of stereo matching, the pixels on the left image are compared against the pixels on the right image along the same horizontal straight line (in the case of rectified images). Therefore, computing the gradient along the vertical direction does not give extra information about the reliability of the disparity map. The output of the gradients on regular images typically consists of edges and sometimes isolated points. After thresholding the horizontal gradients and applying the depth filter, I can get the far-away regions but it would be very hard to see the far-away areas among edges and points. Therefore, in the interest of good visualization, I increased the contour-like data coming from the texture, such that they really form areas, and this is done by lowpass filtering followed by thresholding. Generally, *Sobel* or *Scharr* operators are used to find the gradients of the image because they also apply Gaussian smoothing to reduce the noise. But in this case, I applied the 1-D Sobel operator without Gaussian filtering which is computationally inexpensive, and applied the unnormalized box filter twice. I applied the box filter twice as it effectively uses the triangular filter.

The low-pass filtered horizontal gradient image was converted into a binary mask by applying a threshold that was chosen empirically. This exact value of the threshold is not important as the results are used only for visual inspection and they are not part of the final vision pipeline. Let I be a grayscale input image, I_x be the derivative of image I in the horizontal direction, then the derivative image I_x can be computed by applying a 1D Sobel filter.

$$I_x = Sobel(I) \tag{14.82}$$

Let *box_filter* be a function that does the lowpass filtering using a kernel of size k, and $\hat{I}_x$ be the smoothed derivative of image I, then the smoothed derivative image $\hat{I}_x$ can be obtained by applying the box filter twice.

$$\hat{I}_x = box\_filter(box\_filter(|I_x|, k), k) \tag{14.83}$$

Let $\hat{I}_{th}$ be the threshold for the smoothed derivative, and txt_I be the binary mask for the texture in the image, then the texture mask txt_I is computed by comparing the value of each pixel in smoothed derivative image $\hat{I}_x$ with the threshold

$$txt\_I(x,y) = \begin{cases} 1 & \text{for} \quad \hat{I}_x(x,y) \geq \hat{I_{th}} \\ 0 & \text{for} \quad otherwise \end{cases} \tag{14.84}$$

where, (x, y) are the pixel indices. Similarly, if I_d is the depth image, d_{th} is the depth threshold for the far-away areas, and dep_I be the binary mask for the depth image, then the depth mask dep_I can be obtained by comparing the depth of each pixel with the depth threshold d_{th}.

$$dep\_I(x,y) = \begin{cases} 1 & \text{for} \quad I_d(x,y) \geq d_{th} \\ 0 & \text{for} \quad otherwise \end{cases} \tag{14.85}$$

These two masks are combined together using the logical *and* operator and applied to the left image of the stereo pair to get the faraway region.

Table 14.3: Parameters used in the visualization of the far-away areas

Parameter name	code name	value
Texture threshold	I_{th}	1000
Depth threshold	d_{th}	40 meters
Kernel size for box filter	k_1	3×3
Kernel size for box filter	k_2	5×5

I considered different instants from the recorded dataset to validate the robustness of the disparity maps obtained from the ZED camera. I used two different kernel sizes for the box filter (see table 14.3) for the visualization. Figure 14.9 and Fig. 14.9d present different cases that can occur very frequently during the operation of the boat. In both cases, we can see that the disparity data is able to identify the farthest areas. In Fig. 14.9l, the far-away areas are highlighted even in the low-exposure settings but the disparity data couldn't handle the reflection of the sun on the water but in the normal settings (see Fig. 14.9i), this issue can not be seen. From these results, it can be stated that the disparity map obtained from the ZED camera can be used to find far-away landmarks but in some cases, it doesn't give the best results as we saw in Fig. 14.9l.

14.8 Different methods to compute the 2D displacement for the detected far distant image regions

As mentioned at the beginning of the chapter, I will be discussing three different methods in the following sections that are based on the principle of estimation of rotational angles from far-away areas. The first two methods consider a window in an image that mostly represents the far-away area and then estimate the 2D displacement of this window in the next image using the PhC method. On the other hand, the third method uses 2D displacement of the keypoints within the frames. All the methods use the 2D displacements to compute the rotational angles as mentioned in section 14.5.

The first method takes only one window from the image and uses the distance-weighted gray value profiles to compute the yaw angle only, whereas the second method takes multiple small windows and computes their displacement in the next image using the PhC and then estimates all the rotational angles. Finally, the third method uses the correspondences obtained from the Correspondence Estimator (CorrEst) module to compute all the rotational angles. It should be noted that the first two methods are area-based methods and they do not require the keypoints and their correspondences, therefore, they can be used anywhere in the whole system, whereas the third method requires the correspondences and therefore, it can be used only after the correspondences are estimated. It also implies that area-based methods can be used to pre-estimate the rotational angles (see section 11.2, p.82) but it is not true for the correspondences-based method.

14.9 Rotation estimation using distant gray value profiles

The main principle of the approach is based on the PhC of two 1D signals. Two 1D signals that can represent the information distributed along the horizontal axis of the previous and the current image are extracted. In the frequency domain, the linear translation of these signals can

(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

(j) (k) (l)

(m) (n) (o)

Figure 14.9: In left figures, the original 2k HD color images are shown. The far-away regions obtained using the kernel of size k_1 and k_2 are shown in the middle and right figures respectively.

be measured as the phase change. Therefore, when the car is moving straight (no horizontal shift in the signal), there is zero phase change and when the car turns, a phase change occurs between the signals that correspond to the horizontal translation between the signals due to the yaw change. The whole algorithm will be discussed in detail in the following sections followed by the results and comments.

14.9.1 Preparation of the 1D signals

I took a rectangular patch window from the previous and the current grayscale images. The width of the window is kept equal to the frame width and the height of the window (win_height) is equal to the $win\_height\_ratio * frame\_height$. The window should be aligned with the horizon of the scene because we are mostly interested in the far region but for the KITTI dataset, I assumed that the horizon is aligned with the camera's horizontal axis and passes through the center of the image. Therefore, the image patch is a horizontal strip taken from the center of the image. Similarly, I took the patch windows from the previous and current depth images using the same window alignment and size.

The patches are still in 2D dimension with size $frame\_width \times win\_height$. To convert these 2D signals into 1D signals, I first weighted the grayscale patches with their corresponding depth patches such that the pixels that are far away are given more importance. A hard threshold on the depth patch can be used to prepare a binary weighted mask of zeros and ones where the ones are given to the pixels that have a depth higher than the threshold, however, this totally ignores the close-range pixels which are crucial during the turns in the urban scenarios where the close range pixels are dominating. Therefore, I passed the depth patch to a sigmoid function (see appendix A.3, p.172) to smoothly assign the weights depending on the depth. After the

Algorithm 16 Signal preparation

procedure $get\_signal(img\_patch, depth\_patch)$
  $signal = [0]_{frame\_width \times 1}$
  $depth\_sigmoid\_patch = \textbf{sigmoid}(depth\_patch, sig\_a, sig\_b)$
  for $[col\_img, col\_weight] \in [img\_patch, depth\_sigmoid\_patch]; i = 0$ **do**
    $sig = \dfrac{col\_img \odot col\_weight}{\textbf{sum}(col\_weight)}$
    if $sum(col\_weight) \geq sig\_th$ **then**
      $signal[i] = sig$
  return $signal$

Table 14.4: Parameters for the yaw estimation using weighted gray value profiles.

Parameter	code name	value
Window height ratio	win_height_ratio	0.2
sigmoid a param	sig_a	1
sigmoid a param	sig_b	5
signal threshold	sig_th	0.3

preparation of the sigmoid depth patch, the weighted average of the grayscale patch is computed[1] where the weights are the sigmoid depth mask. The weighted patch is normalized to improve the signal. If the weighted average is below the sig_th, then the value is replaced with zero (see algorithm 16).

14.9.2 PhC between weighted gray value profiles

The PhC has to be performed in the frequency domain, therefore, I applied the *Fast Fourier Transformation (FFT)* on the 1D signals using the *fft* function from the Numpy library. It is recommended to use the signal length that is a power 2 for the faster computation of the FFT, therefore, the signal needs to be cropped from the center. For example, if the signal length is 1030, then after the cropping, it will be 1024 as it is the largest power of 2 that is just before the signal length. After the cropping of both the previous and current 1D signals, the following steps have been taken to compute the phase change, and the algorithm is given in 17.

- Suppressing the edge effects of the signal using the hamming window.
- Computation of the FFT.

Algorithm 17 Phase Computation

procedure $get\_phase(prev\_sig, curr\_sig)$
  $prev\_sig = \textbf{hamming}(prev\_sig)$
  $curr\_sig = \textbf{hamming}(curr\_sig)$
  $prev\_spectrum = \textbf{fft}(prev\_sig)$
  $curr\_spectrum = \textbf{fft}(curr\_sig)$
  $cross\_spectrum = \overline{prev\_spectrum} \odot curr\_spectrum$
  $ph\_th = 1e^{-4} * \textbf{max}(|cross\_spectrum|)$
  $ph = cross\_spectrum/|cross\_spectrum|$
  for $val \in cross\_spectrum; i = 0$ **do** ▷ Normalization
    if $|val| > ph\_th$ **then**
      $cross\_spectrum[i] = val/|val|$
    else
      $cross\_spectrum[i] = 0$
  $varphi = \textbf{ifft}(cross\_spectrum)$ ▷ Inverse FFT
  return $varphi$

- Computation of the cross-spectrum. [2]

[1] In the algorithm 16, the symbol $\odot$ has been used to represent the element-wise multiplication of two arrays

[2] In the mathematical notations, if x is the signal then the complex conjugate of the signal will be denoted by $\overline{x}$

- Dynamic estimation of the threshold value to suppress the small phase changes which are due to the noise in the signal.
- Normalization and filtering of the cross-spectrum.
- Inverse Fourier Transformation using the Numpy library.

14.9.3 Yaw angle estimation using the horizontal shift of the signal

In the previous section, I converted the 1D image signals from two image patches and computed the phase change between them. After the inverse FFT, the resulting 1D signal should have a peak that corresponds to the horizontal displacement of the signal. The signal is circular in nature, therefore, in case of no horizontal shift, half of the peak lies on the left edge of the signal and the remaining half of the peak lies on the right edge (see Fig. 14.10). When there is a horizontal displacement between the two input 1D signals, the peak moves accordingly. If the peak is in the right half of the 1D signal then it implies the rotation of the image patches in the opposite direction and therefore, the horizontal shift is negative and has to be mapped to the left. In other words, the range of the shift in the signal is $[-signal\_length/2, signal\_length/2]$, where the $signal\_length$ is the maximum power of 2.

Let $prev_{kp}$ be the keypoint lying at the center of the previous image, $shift$ be the horizontal shift (in pixels) that the previous keypoint $prev_{kp}$ undergoes, and $curr_{kp}$ be the correspondence of the previous keypoint in the current image, then the corresponding point can be calculated using the shift only.

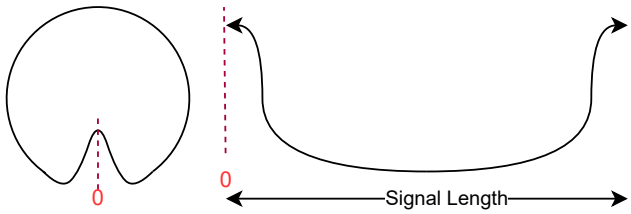


Figure 14.10: The left figure shows the circular nature of the signal after the inverse FFT and the right figure shows when the circular signal is opened to become a 1D signal and the peak splits into two and moves to the edges in case of zero horizontal displacement. The left and right figures are not on the same scale.

$$\begin{aligned} prev_{kp} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ curr_{kp} &= \begin{bmatrix} shift \\ 0 \end{bmatrix} \end{aligned} \tag{14.86}$$

I can safely assume that this keypoint is sufficiently far because we weighted the signal according to the depth. The yaw angle can be computed from the previous $prev_{kp}$ and current keypoint $curr_{kp}$ using eq. 14.87 which is also written here.

$$\theta = \text{atan2}\left(\frac{-x_1'}{f_x}\right) - \text{atan2}\left(\frac{-y_1'}{f_x}\right) \tag{14.87}$$

Algorithm 18 Horizontal shift from the peak

```
1 procedure get_shift(varphi)
2     shift = argmax(varphi)
3     if shift > signal_length/2 then
4         shift = shift − signal_length
      return shift
```

Algorithm 19 Yaw angle pre-estimator

```
1  procedure get_yaw(prev_img, prev_depth_img, curr_img, curr_depth_img)
2      [prev_patch, prev_depth_patch] =
           crop(prev_img, prev_depth_img, win_height_ratio)
3      [curr_patch, curr_depth_patch] =
           crop(curr_img, curr_depth_img, win_height_ratio)
4      prev_sig = get_signal(prev_patch, prev_depth_patch)        ▷ see algorithm 16
5      curr_sig = get_signal(curr_patch, curr_depth_patch)        ▷ see algorithm 16
       // Reduce the signal to have length power of 2.
6      [prev_sig, curr_sig] = crop(prev_sig, curr_sig)
7      varphi = get_phase(prev_sig, curr_sig)                     ▷ see algoritm 17
8      shift = get_shift(varphi)                                  ▷ see algoritm 18
9      [prev_kp, curr_kp] = get_kp(shift)                         ▷ see eq. 14.86
10     yaw = get_yaw_from_corr(prev_kp, curr_kp)                  ▷ see eq. 14.87
       return yaw
```

14.10 Rotation estimation using multiple faraway areas

In the previous section, I considered only a single thin patch at the center of the image and computed the yaw angle by measuring the horizontal shift from the phase change. The method described in this section also uses the PhC to compute the 2D motion between the two frames but it is more similar to the approach mentioned in [2]. In this approach, multiple windows (aligned on the horizontal axis of the image) have been considered over the horizon in the image. The size of the window should be picked in a way such that the largest motion (in pixels) for that window is smaller than half of the window size. It is possible to detect multiple motions within the window using the enhanced PhC ([34]) but this feature is not available in the proposed approach.

Let N be the total number of windows, $w \times h$ be the size of each window patch, $W \times H$ be the size of the image, s be the horizontal spacing between the two windows, hf be the height factor for the placement of windows on the vertical axis of the image, and (x_i, y_i) be the pixel coordinate of the center of the i^{th} window in the image, then the origin of all the windows

Table 14.5: Parameters for the rotation estimation from multiple faraway areas

Parameter description	code name	value
total number of windows (for maritime)	N	45
total number of windows (for KITTI)	N	20
window size	$w \times h$	64 256
spacing between windows	s	40 pixels
height factor	hf	0.48
allowed depth type	d_type_req	*VERY CLOSE*
depth threshold for *FAR* depth type	far_{th}	40 meters
depth threshold for *MEDIUM FAR* depth type	med_far_{th}	20 meters
depth threshold for *CLOSE* depth type	$close_{th}$	5 meters
ratio threshold for the depth type	r	0.5

$(x_i, y_i)\ \forall\ i = 1...N$ can be computed.

$$\begin{aligned} x_i &= \frac{W}{2} - s(\frac{(N-1)}{2} - i - 1) \qquad \forall\ i = 1...N \\ y_i &= H * hf \end{aligned} \tag{14.88}$$

Once the size of the window is defined and the coordinates of the center of the window are computed, N window patches can be extracted from the previous and current images. Unlike the previous method, I used depth data to check if the window is in the far region or not instead of using them as weights. The depth type for each window from the previous image can be classified into four categories (1) *FAR*, (2) *MEDIUM FAR*, (3) *CLOSE*, and (4) *VERY CLOSE*.

Let far_{th}, med_far_{th}, and $close_{th}$ be the depth thresholds for the *FAR*, *MEDIUM FAR*, and *CLOSE* categories respectively, n_far, n_med_far, and n_close be the number of points in the window that is above the far_{th}, med_far_{th}, and $close_{th}$ depth thresholds, r be the ratio threshold which defines the depth type of the window patch, then each window can be labeled into any of the four depth types using the criteria given below.

$$d\_type_i = \begin{cases} \text{FAR, if } n\_far > r(w*h) \\ \text{MEDIUM FAR, if } n\_med\_far > r(w*h) \\ \text{CLOSE, if } n\_close > r(w*h) \\ \text{VERY CLOSE, otherwise} \end{cases} \tag{14.89}$$

The above-mentioned depth types are given a rank in which the *FAR* has the highest rank and the *VERY CLOSE* depth type has the lowest rank. Let d_type_req be the accepted depth type such that the windows with the d_type_req and above this rank are accepted. For example, if the *MEDIUM FAR* depth type is accepted then the windows with *MEDIUM FAR* as well as the windows with *FAR* depth type are accepted but this is not true the other way around. Among the N windows from the previous image, only those windows are processed further that fulfill the depth type filtering. The windows in the current image are also removed if their corresponding window in the previous image didn't survive the depth type filtering.

For the PhC, the Hanning window is applied to all of the windows from the previous and the current image to reduce the edge effects, and the PhC is performed between the corresponding windows using the function from the OpenCV. The function returns a 2D displacement and a score that corresponds to the strength of the identified peak.

Algorithm 20 Rotation estimation from multiple faraway areas

```
1  #DEFINE  SOFT_FACTOR = 0.08
2  procedure get_phc_rotation(prev_img, curr_img, prev_depth_img, win_params)
3      soft_window = generate_hamming_win(win_params) * SOFT_FACTOR
4      w_centers = generate_windows(win_params)                    ▷ see eq. 14.88
5      [prev_w, curr_w, prev_depth_w] =
           extract_windows(prev_img, curr_img, prev_depth_img)
6      for i = 0; i < N; i++ do
7          is_valid = get_depth_type(prev_depth_w(i), d_type_req)  ▷ see eq. 14.89
8          if is_valid then
9              [Δ, scores(i)] = phc(prev_w(i), curr_w(i), soft_window)
10             [points(i), corrs(i)] = create_correspondences(w_centers(i), Δ)
11     roll, pitch, yaw = rpy_estimator(points, corrs, scores)     ▷ see algorithm 21, p.127
       return roll, pitch, yaw
```

The windows from the previous image can be treated as the points (with the center of the windows as their coordinates) that are displaced to a new location in the next image. Let n be the total number of windows in the previous image that survived the depth type filtering, $point_j = (x_j, y_j)$ be j^{th} point in the previous image, $(\partial x_j, \partial y_j)$ be the measured displacement from the PhC, and $corr_j$ be the new location of point $point_j$ in the current frame.

$$corr_j = point_j + (\partial x_j, \partial y_j) \qquad j = 1...n \tag{14.90}$$

The set of points *point* and their displaced correspondences *corr* can be used to estimate the roll, pitch, and yaw using the solution given in section 14.5.

14.11 Rotation estimation from faraway keypoints

This approach is responsible to calculate the rotation estimates from the faraway GFTT and their correspondences. In chapter 12 and chapter 13, we saw how the GFTT keypoints can be generated and their correspondences can be estimated. This approach assumes that the keypoints have been generated and their correspondences are already estimated.

To follow the same principle of using faraway regions to compute the rotation, I first filtered out the keypoints and their correspondences that are not far by comparing their depth with some threshold (d_th). As the number of correspondences is limited in this case and it is possible that none of the correspondence survives the depth filtering, therefore, I reduced the depth threshold by some amount (d_red) at every iteration until the minimum number of correspondences (min_corr) survived or the maximum number of iterations (max_itr) reached. Finally, based on these faraway correspondences, I computed the rotational angles.

14.12 Comparison of different rotation estimation algorithms

I have discussed three methods so far that can compute the rotation from the faraway region. All of these methods use depth to distinguish the faraway region from the close region. Let *Alg 1* be the algorithm described in section 14.9 that can compute only yaw angle, *Alg 2* be the algorithm that uses multiple faraway windows (see section 14.10), *Alg 3* be the algorithm that uses the faraway correspondences (see section 14.11), and *BR* be the algorithm from the [2]. All of these algorithms are run on 11 KITTI sequences that have the ground truth to compare the estimated rotation angles.

Table 14.6 shows the comparison of different algorithms. I computed the mean of the absolute difference between the estimated rotational angle per frame and the true (ground truth) rotation angle per frame for all the frames in the sequence for which I have a valid estimate of the rotation angle.

In some cases, a particular algorithm can not always estimate the rotational angles and therefore, the success rate of the algorithm should be considered as well. Let N be the total number of frames in a sequence, n be the number of frames for which the algorithm returned a valid rotational angle, then the success rate r can be computed by taking the ratio of the n w.r.t the N.

$$r = n/N \tag{14.91}$$

I used the same metric as in [2] to measure the performance of the algorithm. I computed the mean of the absolute difference between the estimated and the true (ground truth) rotation angle per frame for all the n frames. Table 14.6 shows the comparison of different algorithms. In the table 14.6, r is the success rate and val is the mean value for a particular rotational angle. *Alg 1* can not compute the roll and pitch angle, therefore, a cross mark (✗) has been put in front of the roll and pitch for this algorithm. The success rate of *Alg 1* and *Alg 3* for all the sequences is 1, whereas, for the *Alg 2*, it is either 1 or approximately 1. I used the results from the [2] as the benchmark and compared the results of the three proposed algorithms against it. The values in the bold show that the result is better than the *BR* and if it is red in color, then it implies that is best among the three algorithms and if it is bold and red in color then it means that it is the best among all four of the algorithms.

For the roll and pitch, the *Alg 2* outperformed in all the sequences whereas the *Alg 3* gave the best yaw estimates in most of the sequences. The *Alg 1* is a very basic and pre-matured implementation and therefore, it couldn't compete with the other two algorithms but some advanced filtering in the *Alg 1* could have given completely different results. From the table 14.6, the following conclusion can be drawn.

- The keypoints-based rotation estimator gives the best yaw estimates in general but in the case of road sequences such as 01, 04, and 06, the performance of the algorithm drops.
- The multiple far-away windows-based algorithm gives the best roll and pitch estimates as this algorithm approximates the motion of the whole window and reduces the noises in the roll and pitch estimates which have a zero mean value. It is not possible in the keypoints-base rotation estimation as the noise in any correspondence of the keypoint directly influences the estimate of the rotational angle.
- The *Alg 2* also dominated the *Alg 3* for the road sequences (seq 01, 04, and 06) because the motion generated due to the fellow moving vehicles is reduced by considering a window.

Algorithm 21 Rotation estimation from far-away points

procedure $get\_rotation(prev\_kps, curr\_kps, prev\_kps\_depth, corr\_scores)$
  $adap\_d\_th = d_{th}$
  for $i = 0; i < max\_itr; i++$ **do**
    $num\_far\_kps = \sum\{1 \text{ if } d > adap\_d\_th \text{ else } 0 \quad \forall\ d \in \text{prev\_kps\_depth}\}$
    if $num\_far\_kps \geq min\_corr$ **then**
      // Keep far keypoints only.
      $far\_prev\_kps =$ **filter**$(prev\_kps, prev\_kps\_depth, adap\_d\_th)$
      // Keep correspondences of far keypoints only.
      $far\_curr\_kps =$ **filter**$(curr\_kps, prev\_kps\_depth, adap\_d\_th)$
      $roll, pitch, yaw =$ **rpy_estimator**$(points, corrs, scores)$ ▷ see alg. 21, p.127
      return $SUCCESS, roll, pitch, yaw$
    else
      $adap\_d\_th = adap\_d\_th - d\_red$ ▷ Depth threshold reduction
  return $FAILURE, NONE, NONE, NONE$

Table 14.7 and table 14.8 show the sub-sequences from the KITTI sequence 00 in which the yaw error per frame is large when estimated using *Alg 3*. I commented on these sequences to highlight the challenging cases for the rotation estimator.

14.13 Conclusion on the rotation estimation from faraway region

In this chapter, I derived the equations to estimate the rotational angles from the correspondences and then proposed three different algorithms. The first approach used the distance-weighted gray profile to measure the yaw by computing the phase change between two images. This approach is very simple but effective and can be used as a yaw rotation pre-estimator if some filtering techniques are also introduced. The second approach used multiple far-away windows to measure all three rotational angles and it also used the PhC technique. It doesn't use the keypoints as well therefore, it can be used as a rotation pre-estimator but the yaw estimates are somewhat not accurate as the origin method proposed in [2] or the third approach (*Alg 3*). The final approach used the correspondences from the ego-motion sub-system to estimate the rotational angles. It gave good rotation estimates and especially yaw which is the most dominant and important rotational angle.

In an ideal case, the window/area-based algorithm (*Alg 1* or *Alg 2*) should have been used as a rotation pre-estimator only but due to the limited time, I managed to explore the keypoints-based rotation estimation algorithm the most and decided to use it initialize the rotational angles for the final pose estimator. In future work, the window/area-based algorithm should be more focused to use them in the rotation pre-estimator module and the decision whether to keep the *Alg 3* for the rotation initialization or not shall be made.

Table 14.6: Comparison of different methods based on the rotation estimation from the faraway region.

Seq			0	1	2	3	4	5	6	7	8	9	10
Roll	BR	r	0.961	0.991	0.938	0.979	0.947	0.983	0.984	0.976	0.958	0.940	0.934
		val	0.121	0.078	0.159	0.086	0.142	0.108	0.089	0.100	0.104	0.136	0.139
	Alg 1	r	0	0	0	0	0	0	0	0	0	0	0
		val	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Alg 2	r	0.998	0.994	0.999	1	1	1	1	0.996	0.998	1	0.997
		val	**0.054**	**0.046**	**0.054**	**0.034**	**0.051**	**0.044**	**0.040**	**0.043**	**0.052**	**0.061**	**0.060**
	Alg 3	r	1	1	1	1	1	1	1	1	1	1	1
		val	0.131	**0.066**	**0.140**	**0.065**	**0.077**	**0.095**	**0.069**	0.109	0.107	**0.128**	**0.134**
Pitch	BR	r	0.999	0.999	0.999	0.998	0.996	0.999	0.999	0.999	0.999	0.999	0.999
		val	0.044	0.024	0.053	0.038	0.040	0.033	0.026	0.030	0.034	0.040	0.040
	Alg 1	r	0	0	0	0	0	0	0	0	0	0	0
		val	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Alg 2	r	0.998	0.994	0.999	1	1	1	1	0.996	0.998	1	0.997
		val	**0.034**	**0.021**	**0.032**	**0.026**	**0.021**	**0.025**	**0.020**	**0.022**	**0.028**	**0.029**	**0.032**
	Alg 3	r	1	1	1	1	1	1	1	1	1	1	1
		val	0.079	0.056	0.088	0.063	0.107	0.059	0.058	0.062	0.069	0.079	0.077
Yaw	BR	r	0.999	0.999	0.999	0.998	0.996	0.999	0.999	0.999	0.999	0.999	0.999
		val	0.133	0.068	0.126	0.098	0.121	0.114	0.105	0.151	0.123	0.125	0.121
	Alg 1	r	1	1	1	1	1	1	1	1	1	1	1
		val	0.292	0.328	0.400	0.241	0.356	0.271	0.444	0.347	0.279	0.433	0.305
	Alg 2	r	0.998	0.994	0.999	1	1	1	1	0.996	0.998	1	0.997
		val	**0.128**	0.141	0.149	**0.097**	0.144	**0.102**	**0.089**	**0.094**	**0.112**	0.139	0.134
	Alg 3	r	1	1	1	1	1	1	1	1	1	1	1
		val	**0.095**	0.108	**0.119**	**0.077**	0.218	**0.061**	0.122	**0.062**	**0.076**	**0.103**	**0.079**

Table 14.7: Scenes from the KITTI seq. 00 where yaw error per frame is relatively large.

Seq	Frame number		Comment
	Starting	End	
00	99	108	from beginning to half of the turn
00	175	175	A very close point tracked
00	195	108	from beginning to half of the turn
00	347	360	Because of the high pitch in motion vectors?
00	426	423	from beginning to half of the turn
00	440	400	A very close points tracked
00	573	580	from beginning to half of the turn
00	655	655	A very close points tracked
00	737	744	during beginning to half of the turn
00	942	954	from beginning to half of the turn
00	962	962	A very close points tracked
00	990	990	A very close points tracked
00	1045	1045	A very close points tracked
00	1055	1072	A close points tracked from trees and a few of them have wrong correspondences
00	1093	1096	A close points tracked from trees and a few of them have wrong correspondences
00	1027	1032	from beginning to half of the turn
00	1192	1192	Motion vectors in trees
00	1214	1221	Close scene
00	1620	1620	Unidentified reason
00	1761	1800	Mostly keypoints on trees
00	1944	1963	Less keypoints, keypoints on trees? High errors
00	2094	1963	Close scene. Some other issues as well as there are high errors
00	2358	2440	error cluster. close scene, keypoints on trees, wrong correspondences maybe
00	2831	2870	close scene with turn
00	3884	3913	a lot of trees

Table 14.8: Scenes from the KITTI seq. 00 where yaw error per frame is huge.

Seq	Frame number		Comment
	Starting	End	
00	1058	1058	cluster of wrong or close correspondences
00	1955	1970	close scene and unidentified reasons
00	2402	2430	wrong correspondences and close scene
00	2836	2870	a closing scene towards trees
00	2965	3000	wrong correspondences from trees and then turns
00	3240	3277	turning around a close scene
00	3340	3383	turning around a close scene

CHAPTER 15

Estimation of the ego-motion from the correspondences

Contents

In the ego-motion estimation sub-system, I talked about the generation of the keypoints in an image, pre-estimation of the pose change that can be used by the correspondence estimator, and then rotation estimation based on these correspondences. The final step for the ego-motion is to estimate the pose change from the correspondences with the help of the estimated rotation from the far-away correspondences.

During the prediction of the pose (see chapter 11, p.79), I said that the pose prediction module uses the previous pose change for the prediction but I didn't talk about what will happen when the system starts. In this chapter, I will talk about another module that handles the initialization (also referred to as *bootstrapping*) of the system. Another module that will be discussed in this chapter is the estimation of the pose change using the correspondences.

The pose can be estimated from two frames if the correspondence between some of their pixels is given. This pose estimation problem can be solved in multiple ways. In [18], the authors talked about 3 different strategies that can be implemented to estimate the pose change which are also briefly mentioned below.

- **2D-to-2D:** In this approach, the depth data is not used, and the Essential matrix is computed from the correspondences. If the camera calibration matrix is not known, then the Fundamental matrix will be computed. The Essential matrix is decomposed into the rotation matrix and the translation vector. In the absence of the depth data, the scale can not be estimated as the translation vector is not up to the scale.
- **3D-to-3D:** In this approach, the keypoints from both frames are converted into their corresponding 3D points, and then 3D points from one Camera Coordinate Frame (CCF)

are transformed into the second CCF using some unknown transformation matrix and then the error between the corresponding 3D points and transformed 3D points is minimized using some non-linear optimizer.

- **3D-to-2D:** This approach is considered to be the best among the three of them if the depth data is available. It also computes the 3D points from the keypoints from one frame but unlike the previous case, it projects these 3D points on the other frame and then minimizes the reprojection error. It is better than the 3D-to-3D approach because the previous approach triangulates the keypoints twice (one for each frame) and because of that, the error in the second triangulation gives more uncertainty to the pose estimation. Therefore, the minimization of the reprojection error is proven to be more robust than the above methods. This approach is also called Perspective-n-Point (PnP).

15.1 Bootstrapping of the ego-motion estimator

This module handles the initialization of the system for the first two timesteps. Let $\mathbf{T}_0^w$ be the pose of the CCF at first timestep $t = 0$ w.r.t theWorld Coordinate Frame (WCF), then the pose $\mathbf{T}_0^w$ is initialized to be an identity matrix of size 4×4.

$$\begin{aligned} \mathbf{T}_0^w &= identity(4) \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{15.1}$$

When the second frame arrives in the system at timestep $t = 1$, theKeypoints Detector (KptDet) module detects the keypoints in the first frame and the correspondence predictor predicts the location of these keypoints in the second frame but the pose available to the system is the initial pose $\mathbf{T}_0^w$. This initial pose contains no knowledge of the motion and therefore it can not be used to predict the next pose change $\mathbf{T}_0^1$ and later used by the correspondence predictor. In this case, the bootstrapping comes in handy as it estimates the pose change of the CCF $\mathbf{T}_0^1$ between the first $t = 0$ and second timestep $t = 1$.

The bootstrapper uses the description-based (see section 3.5.1, p.3.5.1) pose change estimation to estimate the pose change $\mathbf{T}_0^1$. I extracted the ORB features from the first and second frame using the OpenCV library [1] and then matched them using the *knn* based Brute Force matching[2]. In this matching, the algorithm computes the distance between the descriptor from frame 1 with all the descriptors from frame 2 and keeps the top two matches that have the least distance. The knn approach was used such that the *ratio test* can be applied to reject the outliers after the matching of the features. Let $d_{i,1}$ and $d_{i,2}$ be the distances for the top two matches in the second frame for a i^{th} descriptor in the first frame, N be the total number of detected features in the first frame such that $i = 1...N$, and r_{th} be the ratio threshold, then, in the ratio test, the ratio r of the least distance $d_{i,1}$ is taken w.r.t the second least distance $d_{i,2}$ such that is this ratio r is below the threshold r_{th}, then the match can be accepted.

$$match_i = \begin{cases} \text{True, if } \dfrac{d_{i,1}}{d_{i,2}} < r_{th}, \\ \text{False, otherwise} \end{cases} \qquad \forall\ i = 0...N \tag{15.2}$$

[1] The tutorial to find the ORB features is here.

[2] The link to the tutorial is here.

Once, the correspondences are found between the first two frames, the pose change is computed using the correspondence-based pose estimator which will be discussed in the next section. The pose change $\mathbf{T}_0^1$ estimated by the feature-based ego-motion estimator during the bootstrapping is passed to the pose-change predictor where it doesn't process this pose change and forwards it to the correspondence predictor.

15.2 Pose change estimation using the correspondences

As mentioned at the beginning of the chapter, I had three choices to compute the pose change from the correspondences but I used the PnP based approach because it gives much better results as compared to the other two approaches. In chapter 13, I also mentioned that the outliers in the correspondences are very dangerous for the whole system and they should be identified at all costs. Taking this comment into consideration, I wanted to use a pose estimator that can ignore as much as outliers as possible during the pose change estimation. I decided to use the *solvePnPRansac*[3] function from the OpenCV library. This function uses the Random sample consensus (RANSAC) to filter out the outliers together with the PnP to estimate the pose change.

This function accepts the 3D keypoints from the previous frame and their corresponding 2D points in the current frame. Therefore, the 2D keypoints have to be converted into the 3D points. The eq.13.1 can be used to project the keypoint from the pixel space to the Cartesian space. Let $[x_1', x_2']^T$ be the keypoint in pixel coordinate, d be the depth of the keypoint depth in the frame I_{t-1}, and $[X_1, X_2, X_3]^T$ be the corresponding 3D coordinate of the keypoint w.r.t the Camera Coordinate Frame (CCF) at timestep $t-1$. With this, the keypoint can be projected from the image space to the 3D space.

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = d \begin{bmatrix} \frac{(x_1' - c_x)}{f_x} \\ \frac{(x_2' - c_y)}{f_y} \\ 1 \end{bmatrix} \tag{15.3}$$

Also, the function assumes an identity transformation matrix in case the initial transformation matrix is not provided but in my case, I have the prediction of the pose change (see section 11.1.2, p.81) and the estimation of the rotational angles from the far-away keypoints (see section 14.11, p.125). The pose information obtained from these two can be fused together using algorithm 22 to obtain an initial transformation matrix.

Finally, the 3D points in the w.r.t the previous CCF, their correspondences in the current frame, and the initial estimate of the pose change can be used together to estimate the pose change between the frames.

[3]The link to the tutorial is here.

Algorithm 22 Fusion of pose prediction and rotation from faraway keypoints

```
1 procedure get_init_pose(prev_pose, prev_kps, curr_kps, prev_kps_depth, corr_scores)
2     pred_pose = pose_predictor(prev_pose)                    ▷ see section 11.1.2, p.81
3     status, r, p, y =
          get_rotation(prev_kps, curr_kps, prev_kps_depth, corr_scores)   ▷ see alg. 21.
4     if status == SUCCESS then
5         [R̂, t̂] = decompose(pred_pose)
6         R = euler2rotation(r, p, y)                          ▷ see eq. A.9.
7         fused_pose = compose(R, t̂)
8     else
9         fused_pose = pred_pose
      return fused_pose
```

CHAPTER 16

Stereo Keypoint Matching: An overview

Contents

As a reminder, I have been focusing on ego-motion estimation using the left image from the stereo camera only and using the right image to generate the depth map only. This approach to estimate the pose change between two timesteps is sufficient and can be used without any roadblocks, however, in this approach, a lot of information from the right image has been ignored. When I found the keypoints in the left image, I only tried to find their correspondences in the next left image using the Correspondence Estimator (CorrEst) module (see chapter 13, p.91). The correspondence optimizer provides the final correspondence by minimizing an error metric used by the Lukas-Kanade (LK) differential matcher. In most cases, the optimized correspondences are very close to the true correspondences but in some rare cases where unexpected motion occurs, a few outliers in the optimized correspondences can be observed. These outliers have a drastic effect on the ego-motion estimation and therefore, some measures should be taken to minimize the number of outliers.

Stereo keypoint matching is the extended version of the previous approach and can be implemented only for setups that have more than one camera, such as a stereo camera, with overlapping images. It imposes more constraints while doing the correspondence estimation such that reliable correspondences can be estimated.

The idea of the stereo keypoint matching algorithm is to find the keypoints in the left image (using a keypoint extractor) and then find the corresponding keypoints in the right image of the same stereo-pair (using the disparity image) and then find their correspondences in the left and right image of the next stereo-pair respectively by minimizing a joint loss function. In an ideal case, the correspondences found in the left and right image for an original keypoint from the previous left image should lie on the same horizontal axis (if the stereo images are rectified).

The joint optimization of the correspondences can be done in Block Matching (BM) mode or in differential LK style. The former is simple to implement and the latter is much more difficult to implement.

In this chapter, I will discuss the correspondence predictor that can be used by a stereo-matching algorithm followed by a loss function for the pose change estimation. It should be noted that the joint optimization of the correspondences and the pose change estimation are left for future work and this algorithm has no influence on the ego-motion estimation sub-system that I proposed before.

16.1 Stereo Correspondence Prediction

The stereo correspondence prediction is similar to the correspondence predictor proposed for the ego-motion estimation sub-system (see section 13.1, p.92) but the former also predicts the location of the keypoints in the right image of the stereo pair (see Fig. 16.1). This process is done in three steps.

- The keypoint in the previous left image is predicted in the current left image.
- The keypoint in the previous left image is predicted in the previous right image.
- The keypoint in the previous right image is predicted in the current right image.

16.1.1 Prediction of keypoints from left-to-left image

The derivation for the prediction of the keypoint on the next frame using an estimate of the pose change already has been done in section 13.1, p.92 but I derived it again in this section because of the new notations used for the stereo pair (see Fig. 16.2).

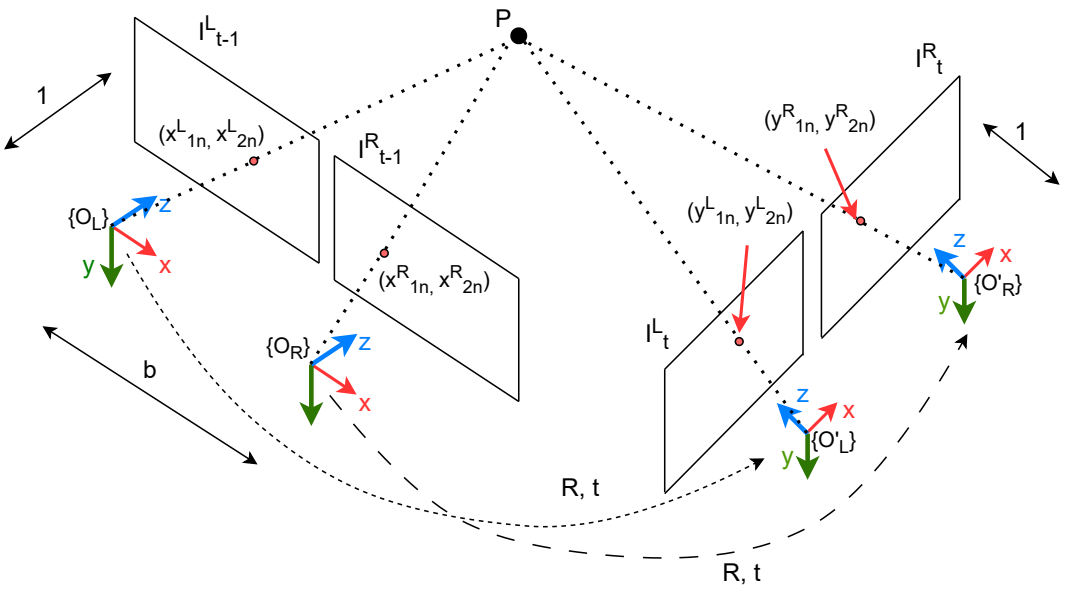


Figure 16.1: Projection of a point P on two stereo images in two different positions

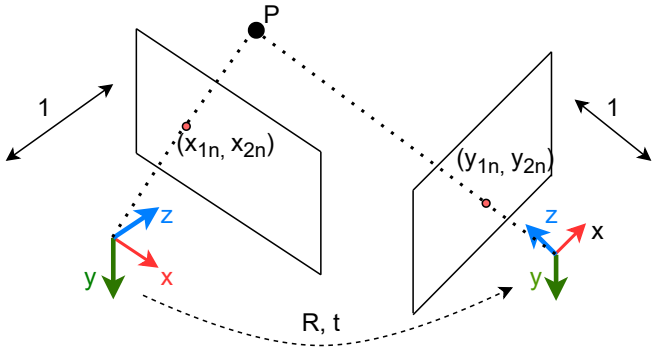


Figure 16.2: Prediction of keypoint location

Let I^L_{t-1} be the left image from the stereo pair at timestep $t-1$, $\vec{x}^L_n \overset{def}{=} \begin{bmatrix} x^L_{1n} & x^L_{2n} \end{bmatrix}^T$ be the normalized coordinate vector of the keypoint from left image I^L_{t-1}, and d^L be the depth of the keypoint and $\vec{x}^L$ be the 3D point in left CCF at timestep $t-1$, then the normalized coordinate vector $\vec{x}^L_n$ can be mapped to the 3D point $\vec{x}^L$ using the depth d^L.

$$\vec{x}^L = \begin{bmatrix} x^L_1 \\ x^L_2 \\ x^L_3 \end{bmatrix} = d^L \begin{bmatrix} x^L_{1n} \\ x^L_{2n} \\ 1 \end{bmatrix} \tag{16.1}$$

Let $\mathbf{T}^t_{t-1}$ be the prediction of the pose change (see chapter 11, p.79) of the CCF from timestep $t-1$ to timestep t, then it can be decomposed into the rotation matrix $\mathbf{R}$ and the translation vector $\vec{t}$.

$$\mathbf{T}^t_{t-1} = \begin{bmatrix} \mathbf{R} & \vec{t} \\ 0_{1\times 3} & 1_{1\times 1} \end{bmatrix} \tag{16.2}$$

Let $\vec{y}^L$ be the transformed 3D point $\vec{x}^L$ w.r.t the CCF at timestep t, then the same transformation matrix $\mathbf{T}^t_{t-1}$ can be used to transform the 3D point $\vec{x}^L$ from the CCF at timestep $t-1$ to the CCF at timestep t.

$$\vec{y}^L = \begin{bmatrix} y^L_1 \\ y^L_2 \\ y^L_3 \end{bmatrix} = \mathbf{R}\vec{x}^L + \vec{t} \tag{16.3}$$

In the homogeneous coordinates,

$$\begin{bmatrix} \vec{y}^L \\ 1 \end{bmatrix} = \mathbf{T}^t_{t-1} \begin{bmatrix} \vec{x}^L \\ 1 \end{bmatrix} \tag{16.4}$$

Let $\vec{y}^L_n$ be the normalized coordinates of the projection of the 3D coordinate y^L on the left image I^L_t at timestep t, then the predicted keypoint $\vec{y}^L_n$ can be computed from the 3D point $\vec{y}^L$.

$$\vec{y}^L_n = \begin{bmatrix} \vec{y}^L_{1n} \\ \vec{y}^L_{2n} \end{bmatrix} = \frac{1}{y^L_3} \begin{bmatrix} y^L_1 \\ y^L_2 \end{bmatrix} \tag{16.5}$$

16.1.2 Prediction of keypoints from left-to-right image

Let I^R_{t-1} be the right image from the stereo pair at timestep $t-1$, and $\vec{x}^R_n \overset{def}{=} \begin{bmatrix} x^R_{1n} & x^R_{2n} \end{bmatrix}^T$ be the normalized coordinate vector of the keypoint $\vec{x}^L_n$ in right image I^R_{t-1}, f_x be the focal length of the camera in the horizontal axis, and d^L be the disparity between the keypoint in the left and right image of the stereo pair w.r.t, then the keypoint in the right image $\vec{x}^R_n$ can be estimated from the keypoint in the left image $\vec{x}^L_n$ using the disparity d^L.

$$\vec{x}^R_n = \begin{bmatrix} \vec{x}^R_{1n} \\ \\ \vec{x}^R_{2n} \end{bmatrix} = \begin{bmatrix} \vec{x}^L_{1n} \\ \\ \vec{x}^L_{2n} \end{bmatrix} + \begin{bmatrix} \dfrac{d^L}{f_x} \\ \\ 0 \end{bmatrix} \tag{16.6}$$

16.1.3 Prediction of keypoints from right-to-right image

In the case of the rectified stereo images, the depth of any 3D remains the same whether it is w.r.t the left CCF or w.r.t the right CCF. Let $\vec{x}^R$ be the 3D point for the keypoints $\vec{x}^R_n$ in the right CCF at timestep $t-1$, then the 3D point $\vec{x}^R$ can be computed using the depth d^L and the normalized image coordinates of the keypoint $\vec{x}^R_n$.

$$\vec{x}^R = \begin{bmatrix} x^R_1 \\ \\ x^R_2 \\ \\ x^R_3 \end{bmatrix} = d^R \begin{bmatrix} x^R_{1n} \\ \\ x^R_{2n} \\ \\ 1 \end{bmatrix} \tag{16.7}$$

The 3D point $\vec{x}^R$ can not be pre-multiplied with the transformation matrix T^t_{t-1} directly to get the transformed 3D point $\vec{y}^R$ as in the section 16.1.1. For the left camera, it was possible because the transformation matrix T^t_{t-1} and the 3D point $\vec{y}^L$ were both in the left CCF. For the projection of points in the right image, the transformation between the left camera and the right camera has to be considered as well.

As the left and the right camera of the stereo setup are fixed on a rigid body, the transformation between them always remains constant (assuming the impact of the temperature, vibrations, tear, and wear on the stereo rig is negligible). In our case, the images are already rectified so the only motion parameter for the transformation between the left and the right camera is the baseline b along the x-axis of the CCF. Let $\mathbf{T}^R_L$ be the transformation matrix that transforms the point from the left CCF to the right CCF, then the transformation matrix $\mathbf{T}^R_L$ depends only on the baseline b.

$$\mathbf{T}^R_L = \begin{bmatrix} 1 & 0 & 0 & -b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{16.8}$$

Let $\mathbf{T}^L_R$ be the inverse of the transformation matrix $\mathbf{T}^R_L$.

$$\mathbf{T}^L_R = \begin{bmatrix} 1 & 0 & 0 & b \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{16.9}$$

To transform the 3D point $\vec{x}^R$ from the right CCF at time step $t-1$ to the right CCF at time step t, it is first premultiplied with $\mathbf{T}^L_R$ to transform it to left CCF, then premultiplied with $\mathbf{T}^t_{t-1}$

to transform it to left CCF at time step t and then finally premultiplied with $\mathbf{T}_L^R$ to transform it to the right CCF at timestep t. The order of the transformations is summarized below.

$$\begin{bmatrix} \vec{y}^R \\ 1 \end{bmatrix} = \mathbf{T}_L^R \mathbf{T}_{t-1}^t \mathbf{T}_R^L \begin{bmatrix} \vec{x}^R \\ 1 \end{bmatrix} \tag{16.10}$$

Finally, the 3D coordinate $\vec{y}^R$ can be projected back onto the right image plane I_t^R, and the normalized coordinates y_n^R can be obtained.

$$y_n^R = \begin{bmatrix} y_{1n}^R \\ \\ y_{2n}^R \end{bmatrix} = \frac{1}{y_3^R} \begin{bmatrix} y_1^R \\ \\ y_2^R \end{bmatrix} \tag{16.11}$$

16.1.4 Comments on the stereo correspondence predictor

The maths behind the correspondence predictor is very straightforward and should work (theoretically) given the accurate depth of the keypoint and the pose change between the two timesteps but in practice, this information is not accurately known to us. The depth is approximately known and the pose change information can only be predicted up to a certain accuracy. To check the reliability of this predictor, I tested this algorithm on the KITTI dataset where the transformation from the ground truth can be used and the depth can be calculated using the SGBM method.

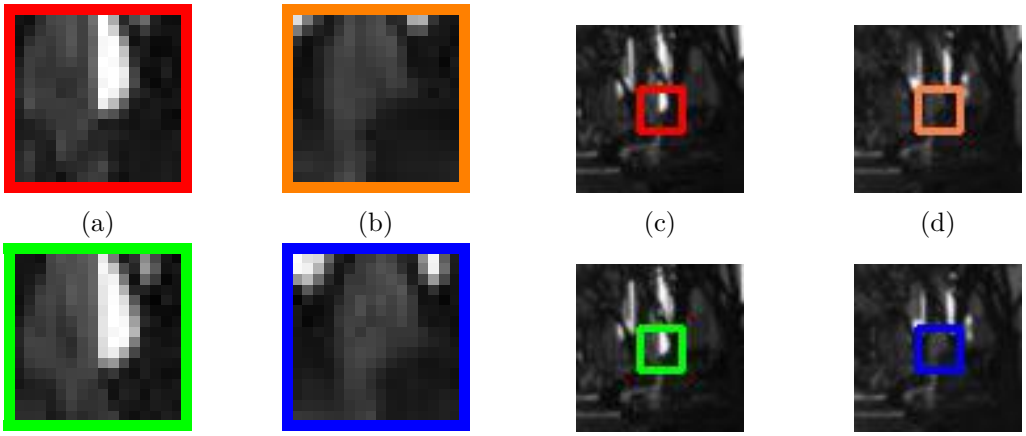
Fig. 16.3 shows the results of the stereo correspondence predictor on the KITTI dataset. Fig. 16.3a reflects the patch of size 16×16 around a GFTT keypoint in the left image and Fig. 16.3b shows the corresponding patch in the right image obtained using the disparity of the GFTT keypoint. Patches with size 64×64 are shown in Fig. 16.3c and Fig. 16.3d for better visualization of 16.3a and 16.3b respectively. The predicted location of the keypoint at the next time step on the left and right images are shown in Fig. 16.3e and 16.3f respectively. Fig. 16.3g and 16.3h shows fig. 16.3e and 16.3f but with bigger patch size for better visualization. From Fig. 16.3e and Fig. 16.3f, it can be said that the predictor did its job well and gave a good approximate of the location where this keypoint can be found in the next stereo frames. However, it should be noted that it failed sometimes to predict the location of the keypoint with good accuracy. This behavior could be due to poor disparity estimation.

16.2 Loss function for the motion estimation from stereo correspondences

In the previous section, I talked about the stereo correspondence predictor that can predict the location of the GFTT keypoint from the previous left image to the previous right image and the current stereo images. The correspondences can be optimized if I have an optimizer that slides the search window in the current left and right images together such that combined error metrics (for say SSD) can be minimized. Let SSD be a function that computes the SSD between the two patches and ∂ be the 2D displacement that needs to be estimated to optimize the correspondences, then the loss function for the stereo correspondence optimizer may look like the following.

$$\min_{\partial} \left\{ SSD((y_n^L + \partial), x_n^L) + SSD((y_n^R + \partial), x_n^R) \right\} \tag{16.12}$$

For the pose estimation from the stereo correspondences, we can combine the reprojection error from the correspondences from the left frames and the right frames respectively. Let



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 16.3: The results from the stereo correspondence detector. The images are explained in the section 16.1.4.

$\hat{\mathbf{T}}_{t-1}^{t}$ be the homogeneous transformation matrix that defines the pose change between two timesteps, *reproj(*$\mathbf{T}$*, x)* be a function that transforms 3D point x to some other reference frame using homogeneous transformation matrix $\mathbf{T}$ (see eq. 16.4) and then project it on the image at timestep t using eq. 16.5. With this, the pose changes $\hat{\mathbf{T}}_{t-1}^{t}$ can be optimized such that the joint reprojection error is minimized.

$$\min_{\hat{\mathbf{T}}_{t-1}^{t}} \left\{ ||y_n^L + \partial - reproj(\hat{\mathbf{T}}_{t-1}^{t}, x^L)|| + ||y_n^R + \partial - reproj(\mathbf{T}_L^R \hat{\mathbf{T}}_{t-1}^{t} \mathbf{T}_R^L, x^R)|| \right\} \tag{16.13}$$

Part V

Experiments and Results

CHAPTER 17

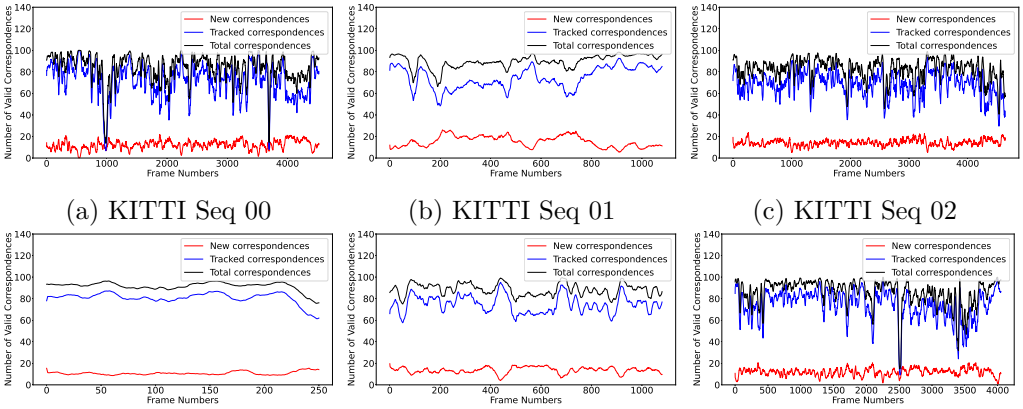
Incremental improvement of the visual odometry pipeline

Contents

In the previous chapters, I discussed the main architecture of the ego-motion sub-system (see chapter 5, p.31) and the corresponding modules and the components of the sub-system (see part IV, p.73). The final module of the ego-motion estimation sub-system returns the pose change between two timesteps. It is crucial to check the performance of the sub-system against the ground truth such that the edge cases can be identified and the performance of the system can be improved. If any of the modules is not carefully designed or the parameters are not carefully tuned, then the accuracy can be dropped significantly. In this chapter, I have looked into the various aspects of the system and tried to identify the issues in the system and what can be reconfigured to make the system better.

17.1 Analysis of the correspondences estimator

A good and robust Correspondence Estimator (CorrEst) should find the accurate correspondences and maintain the total number of correspondences. In the absence of true correspondences, the correspondence can be considered good only if the residual error between the corresponding keypoint and the original keypoint is below some threshold (see algorithm 14, p.95). If the CorrEst couldn't find enough correspondences, then the pose can not be estimated. For example, the *5 Point algorithm* ([33]) requires at least 5 correspondences to estimate the pose.



(a) KITTI Seq 00 (b) KITTI Seq 01 (c) KITTI Seq 02

(d) KITTI Seq 04 (e) KITTI Seq 06 (f) KITTI Seq 08

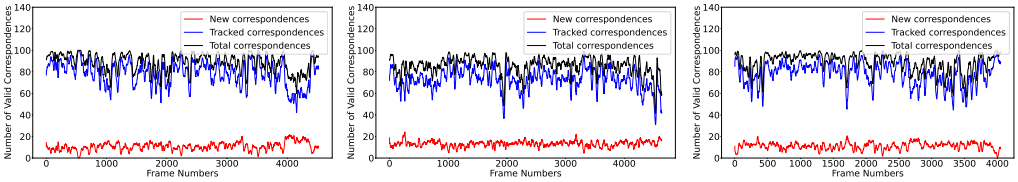
Figure 17.1: The moving average on the different total valid correspondence statistics for different KITTI sequences without pyramid approach. The red and blue plot shows the number of valid correspondences for newly generated keypoints and previously tracked keypoints. The black plot shows the total number of valid correspondences

Therefore, I decided to analyze the CorrEst module by looking at the total number of correspondences. The total number of correspondences consists of two types of correspondences. First, the correspondences of the previously tracked keypoints and the second from the newly generated keypoints. In chapter 12, I mentioned that the Keypoints Detector (KptDet) module was able to detect approximately the maximum number of keypoints *max_total_kps* (including the previously tracked keypoints and the new keypoints), the valid number of correspondences between any pair of frames can't exceed the maximum number of keypoints i.e. *max_total_kps*.

Figure 17.1 shows the number of valid correspondences (after the moving average) for different KITTI sequences. When I ran the CorrEst module on seq. 00, I found out that the number of valid correspondences was dropping below half of the maximum allowed correspondences during the sharp turns (refer fig. 17.1a). To validate this observation, I ran the CorrEst module on sequence 01 in which the car takes the turn but with a bigger turning radius, sequence 04 in which the car takes no turn at all, sequence 06 in which the car takes two 180° turns on the road, and finally sequences 02 and 08 which are similar to sequence 00 in terms of the sharp turns of the car. The sudden drop in correspondences can be seen in sequences such as 02, and 08 as well as shown in Fig. 17.1c and Fig. 17.1f respectively. Sequence 01 from the KITTI dataset is a highway sequence and the car didn't take any sharp turn on the highway which is the most suitable condition for the current LK matcher settings but for the other urban sequences such as 00, 02, and 08, I found that the success rate for CorrEst to find the correspondences decreased during the sharp turns.

From fig. 17.1b, fig. 17.1d, and fig. 17.1e, it is clear that the module works very well when the turns have a high radius of curvature but on the other hand, the performance decreases significantly when the ego-vehicle takes any sharp turns (see fig. 17.1a, fig. 17.1c, and fig. 17.1f). The possible reason behind this limitation of the module is the inaccurate prediction of the correspondences. When the vehicle was accelerating or deaccelerating during sharp turns, the previous pose used by the correspondence predictor underestimated and overestimated the turns respectively resulting in invalid correspondences.

The possible solutions to improve the performance of the CorrEst module is either by intro-



(a) KITTI Seq 00 (b) KITTI Seq 02 (c) KITTI Seq 08

Figure 17.2: The moving average on the different total valid correspondence statistics for different KITTI sequences with pyramid approach.

ducing a good pose estimator or by letting the Lukas-Kanade (LK) matcher to wider its search area. The former solution is inevitable and should be picked in any case but for the robustness of the CorrEst module, I decided to introduce a pyramid level during the search for the correspondences. This solution proposed no harm or advantage to the sequences 01, 04, and 06, however, the success rate for the valid correspondences got better for the urban sequences (refer fig. 17.2).

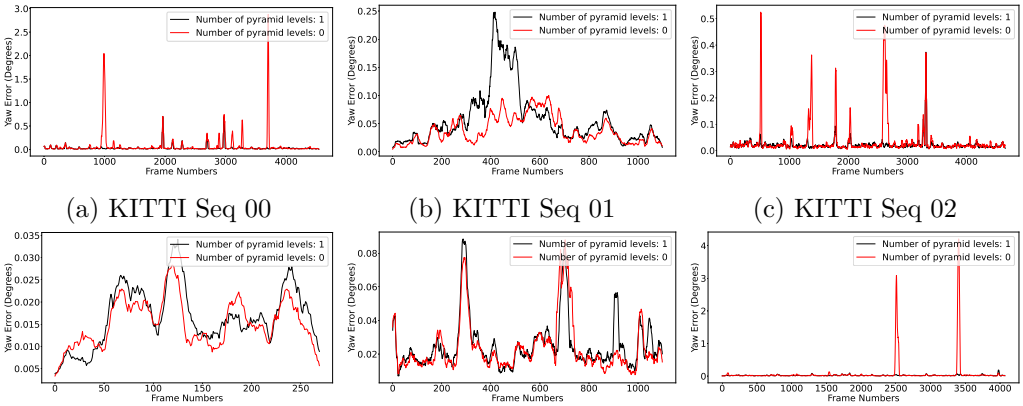
17.1.1 Effect of pyramidal approach on yaw

Previously, I mentioned that the introduction of the one pyramid level in the LK matcher, improved the number of valid correspondences but it doesn't imply that the precision of these correspondences is improved as well. The only way to know this is to look at the error between the estimated pose change and the true pose change (from the ground truth). As we saw before, the CorrEst module is very sensitive to sharp turns, it can be stated that the error in yaw per frame decreases if the pyramid level improved the module. Therefore, I focused on the yaw error per frame only to analyze the performance of the module before and after the pyramidal approach because the yaw is the dominant rotational term for a vehicle and it is also responsible for the drop of the correspondences during sharp turns. If the pyramidal approach reduces the yaw error or at least does not increase it then we can safely assume that there is no side-effect on the rest of the pipeline.

I compared this estimated yaw angle per frame (est_ψ) with the ground truth (gt_ψ) and computed the yaw error per frame ($error_\psi$) by taking their absolute difference.

$$error_\psi = |gt_\psi - est_\psi| \tag{17.1}$$

In figure 17.3, we can see the impact of the introduction of one pyramid level on the yaw error per frame especially for sequences 00, 02, and 08. The difference between the $error_\psi$ for the pyramid and for the non-pyramid approach is so huge for sequences 00, 02, and 08 that the $error_\psi$ appears to be approximately zero as compared to the non-pyramid approach (see Fig. 17.3a, Fig. 17.3c, and Fig. 17.3f). The pyramid approach increases the search area without increasing the search window size and therefore, it can compensate for the overestimation and underestimation of the prediction of the pose change which is required during the sharp turns in the urban sequences. However, the pyramid approach didn't reduce the $error_\psi$ at all for sequences 01, 04, and 06 as shown in Fig. 17.3b, Fig. 17.3d, and Fig. 17.3e respectively. It is due to the fact that the radius of curvature was already big in sequences 01, 04, and 06 as compared to the other remaining sequences. In other words, the pyramid level should be used only if the pose change pre-estimator is not good enough and the sudden change in motions can be observed. In the following sections, I used one pyramid level for all the sequences.



(a) KITTI Seq 00 (b) KITTI Seq 01 (c) KITTI Seq 02

(d) KITTI Seq 04 (e) KITTI Seq 06 (f) KITTI Seq 08

Figure 17.3: The moving average on the absolute yaw error per frame was plotted for different KITTI sequences with and without the pyramid approach. The black and red plots show with and without pyramid level approach.

17.2 Planar pose analysis

The accuracy in the pose or the trajectory itself is very crucial for the ego-motion estimation sub-system. If I ran the system on the whole KITTI sequence, I will obtain a trajectory that may or may not align with the actual trajectory obtained from the ground truth data because the errors in the estimated poses accumulate over time and the trajectory diverges from the ground truth eventually. In this section, I focused on the strong frame-to-frame pose errors first to identify the critical situations in which the system is underperforming.

For the car moving on the planar surfaces, we are mostly interested in the position of the car in the $x-z$ plane and its heading (yaw) in the Camera Coordinate Frame (CCF). It is because the pose in $y-axis$ has a zero mean value because of the planar constraint between the ground and the car and the roll and pitch angles also have a zero mean value as the main source of the roll and pitch angles are the vibrations from the ego-vehicle and the bumps on the road surfaces. Therefore, these three pose parameters (position of the ego-vehicle in $x-z$ plane and its heading (yaw)) have been considered to analyze my ego-motion estimation system. I compared the estimated absolute pose with the ground truth. Instead of comparing the frame-to-frame poses for each timestep, I compared the relative pose of the CCF at timestep at $t = n+L$ relative to the pose at timestep $t = n$ (referred as $\mathbf{T}^n_{n+L}$), where L is the sliding window's width, to check how much the estimated pose diverged since timestep $t = n$ until $t = n + L$. It highlights the critical sequences where the strong pose errors are occurring.

17.2.1 Estimation of the relative pose errors between two timesteps

To compute the relative pose of the camera after L frames w.r.t the current frame n, I need the absolute poses at these timesteps i.e. $\mathbf{T}^w_n$ and $\mathbf{T}^w_{n+L}$ which can be calculated from the relative pose estimates as mentioned in the section 2.1.1. The pose $\mathbf{T}^n_{n+L}$ can be calculated by pre-multiplying the inverse of $\mathbf{T}^w_n$ (see eq. 2.4) with $\mathbf{T}^w_{n+L}$.

$$\mathbf{T}^n_{n+L} = inv(\mathbf{T}^w_n)\mathbf{T}^w_{n+L} \tag{17.2}$$

Using the above equation, I obtained two relative poses: one for the estimated pose data ($est\_\mathbf{T}^{n}_{n+L}$) and the second for the ground truth pose data ($gt\_\mathbf{T}^{n}_{n+L}$). To compute the position error and the orientation error between them, I need to decompose the relative poses or the homogenous transformation matrices into the rotation matrix and the translation vector.

$$\begin{aligned} est\_\mathbf{T}^{n}_{n+L} &= \begin{bmatrix} est\_\mathbf{R}^{n}_{n+L} & est\_\vec{t}^{\,n}_{n+L} \\ 0_{1\times 3} & 1_{1\times 1} \end{bmatrix} \\ gt\_\mathbf{T}^{n}_{n+L} &= \begin{bmatrix} gt\_\mathbf{R}^{n}_{n+L} & gt\_\vec{t}^{\,n}_{n+L} \\ 0_{1\times 3} & 1_{1\times 1} \end{bmatrix} \end{aligned} \tag{17.3}$$

The rotation matrices $est\_\mathbf{R}^{n}_{n+L}$ and $est\_\mathbf{R}^{n}_{n+L}$ can be decomposed into the Euler angles roll ψ, pitch ϕ, yaw θ (see algorithm 23, p.174).

$$\begin{aligned} est_\phi, est_\theta, est_\psi &= rotation2euler(est\_\mathbf{R}^{n}_{n+L}) \\ gt_\phi, gt_\theta, gt_\psi &= rotation2euler(gt\_\mathbf{R}^{n}_{n+L}) \end{aligned} \tag{17.4}$$

Let e_θ be the heading error, then it can be estimated by taking the absolute difference between the estimated and true heading.

$$e_\theta = |est_\theta - gt_\theta| \tag{17.5}$$

Let $est\_t \overset{def}{=} \begin{bmatrix} est\_\vec{t}^{\,n}_{n+L}[0] & est\_\vec{t}^{\,n}_{n+L}[3] \end{bmatrix}^T$ be the estimated relative planar position, $gt\_t \overset{def}{=} \begin{bmatrix} gt\_\vec{t}^{\,n}_{n+L}[0] & gt\_\vec{t}^{\,n}_{n+L}[3] \end{bmatrix}^T$ be the true relative planar position between the timesteps n and $n+L$, and Δ_t be the difference in the estimated and true relative positions, then the error e_t in position can be computed as follows.

$$\Delta_t = est\_t - gt\_t \tag{17.6}$$

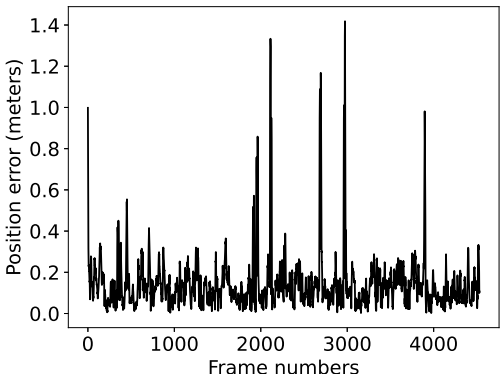
$$e_t = \sqrt{\Delta_t^T \cdot \Delta_t} \tag{17.7}$$

17.2.2 Discussion on the relative pose errors for different KITTI sequences

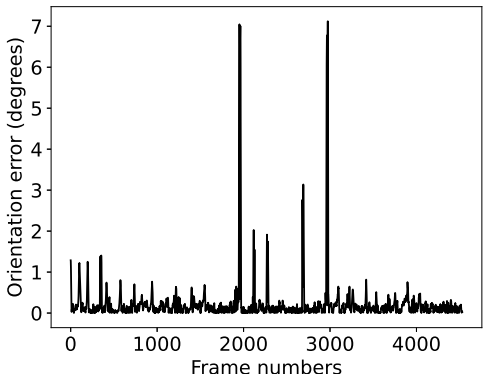
Figure 17.4 and Fig. 17.5 shows the position errors and the heading errors for sequences 00, 01, 02, 04, 06, and 08 with the sliding window length $L = 10$. To compare different sequences w.r.t their position and heading errors, the vertical and the horizontal scale for each plot should have been the same but I decided not to do that because, in some sequences, the heading error and the position error are so huge that these errors in other sequences will not look comparable. Therefore, the reader is advised to look at the range of the vertical and horizontal axis of the plots carefully.

From the Fig. 17.4b, Fig. 17.4f, and Fig. 17.5f we can clearly see that the orientation errors are dominating in sequences 00, 02, and 08 respectively, whereas, the sudden increase in the position error can be spotted in Fig. 17.4c. The sequences 00, 02, and 08 are similar in terms of the objects such as buildings, cars, trees, etc, present in the scene as well the nature of driving (in the urban areas). Therefore, I put my focus on seq. 00, as it has the highest position and orientation errors, and on seq. 01 which has a huge position error for a duration of ≈ 20 seconds.

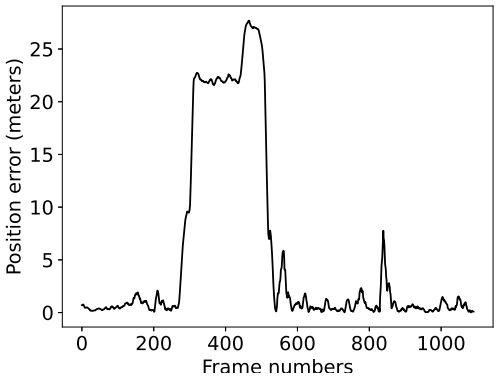
The seq. 01 is a highway sequence where the ego-vehicle is being driven at higher speeds and accompanied by other fellow cars. It is safe to assume that any deviation in the heading of the ego-vehicle corresponds to sudden and huge position errors. It should be noted that these results are generated using the pyramidal approach by the correspondence optimizer. I plotted



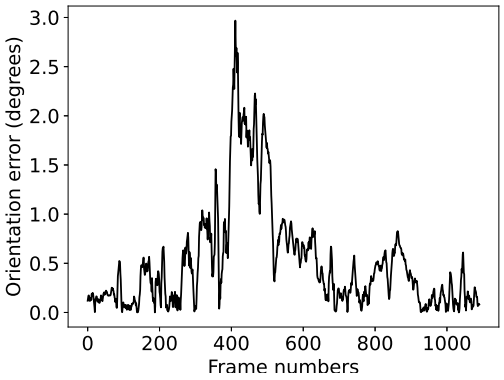
(a) Seq. 00: Position error



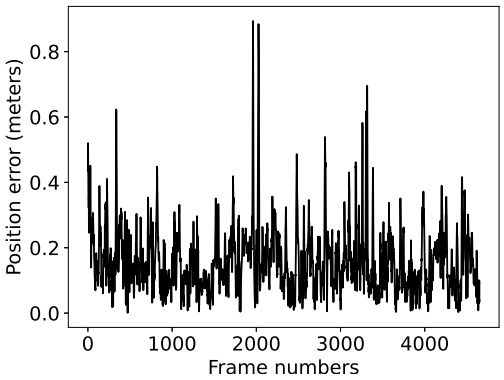
(b) Seq. 00: Heading error



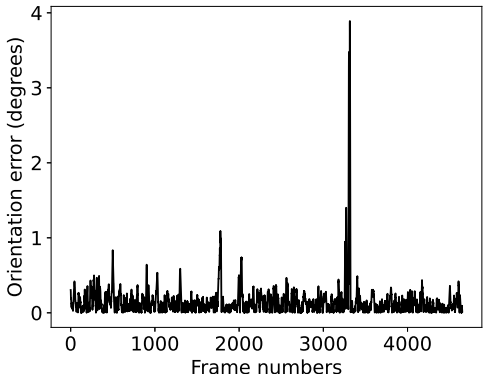
(c) Seq. 01: Position error



(d) Seq. 01: Heading error

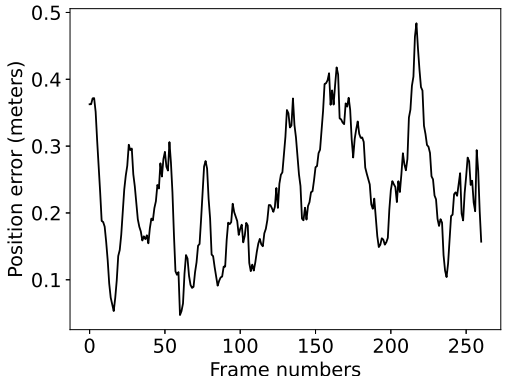


(e) Seq. 02: Position error

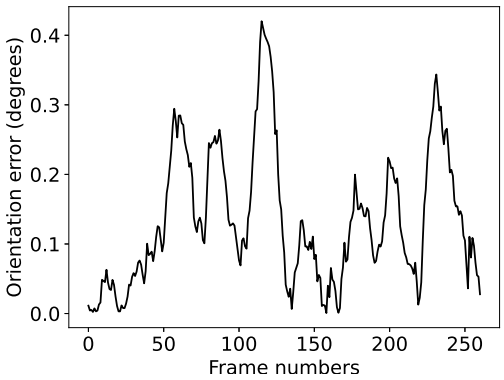


(f) Seq. 02: Heading error

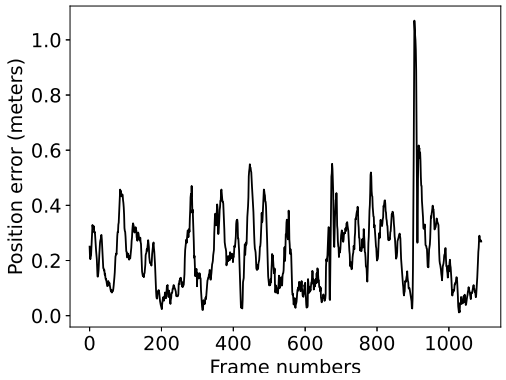
Figure 17.4: The top, middle, and bottom row shows the position errors and orientation errors for KITTI seq. 00, 01, and 02.



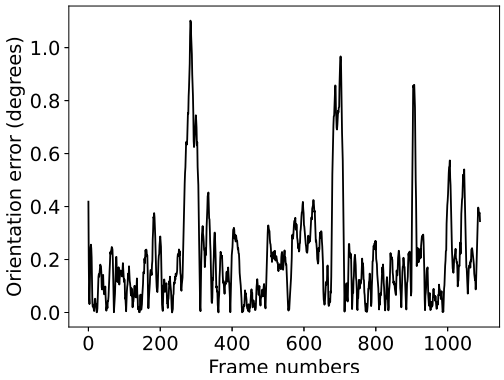
(a) Seq. 04: Position error



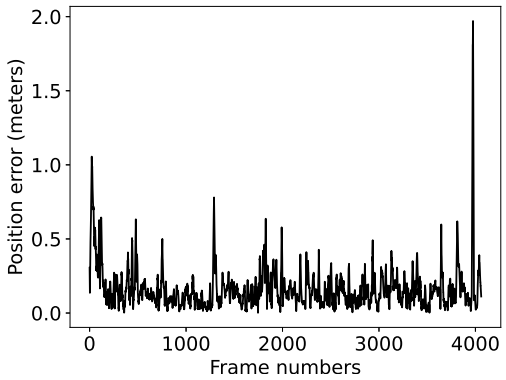
(b) Seq. 04: Heading error



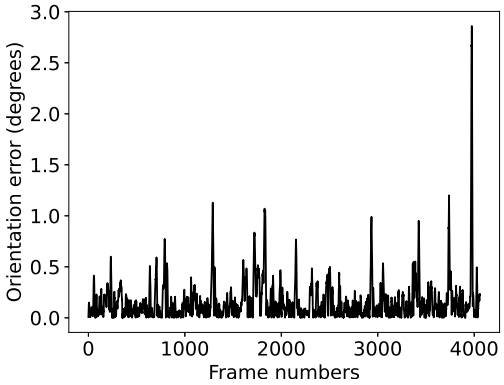
(c) Seq. 06: Position error



(d) Seq. 06: Heading error

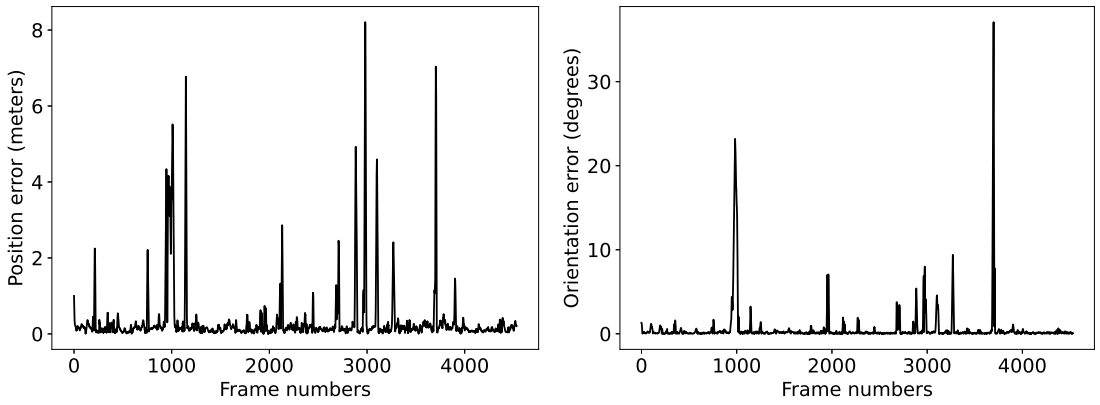


(e) Seq. 08: Position error



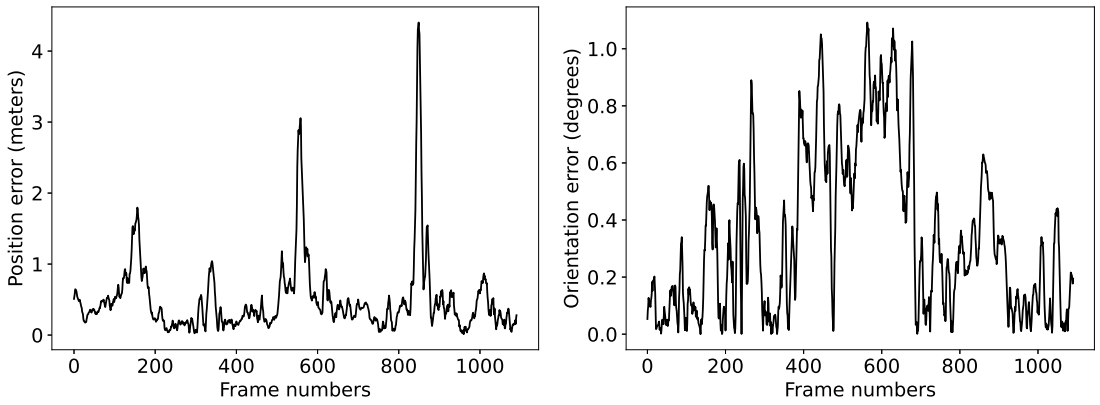
(f) Seq. 08: Heading error

Figure 17.5: The top, middle, and bottom row shows the position errors and orientation errors for KITTI seq. 04, 06, and 08.



(a) Seq. 00: Position error

(b) Seq. 00: Heading error



(c) Seq. 01: Position error

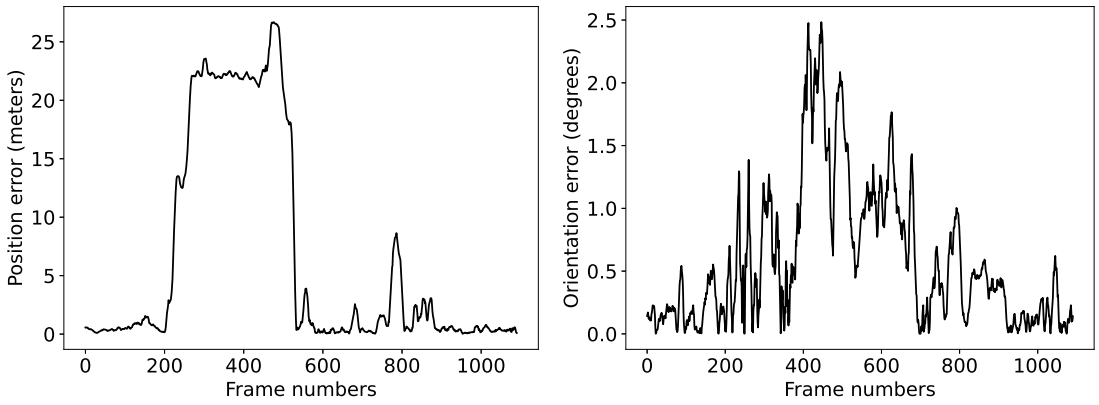
(d) Seq. 01: Heading error

Figure 17.6: The position and the heading errors for KITTI Sequences without using the pyramid levels.

the position and heading error for seq. 00 and 01 for the non-pyramidal approach (the number of pyramid levels is zero) in Fig. 17.6. As mentioned before, the impact of the introduction of the pyramid level on sequence 00 is advantageous whereas, for seq. 01, it made the results more worst. From all of these results, it can be concluded that there the selection of the number of pyramid levels should be done on the basis of the type of driving scenario. In the case of sharp turns, the pyramid levels can improve the correspondence estimation otherwise not.

17.3 The influence of other moving objects on the pose estimation

It can be assumed that the huge position errors in seq. 01 are happening due to the moving cars as the current system has no means to differentiate between a moving and stationary object and therefore, it is quite possible that the keypoints and their correspondences may lie on the moving vehicle which results in the wrong pose estimation. Therefore, I decided to mask out the



(a) (b)

Figure 17.7: The position errors and orientation errors for KITTI seq. 01 fellow vehicles are masked out using object detection.

fellow vehicles with the help of an object mask using YOLO-v8 (see section 7.4, p.50) such that no keypoint lie on the fellow cars irrespective of the motion. Figure 17.7 shows the position and heading errors when the object mask was used along with one pyramid level on seq. 01. The maximum heading error dropped from 3 degrees to 2.5 degrees in the presence of object masking (see Fig. 17.4d and Fig. 17.7b) but the position error didn't change much.

Also, even in the presence of the object detection mask, the influence of moving objects from the ego-motion estimation can not be removed completely. In Fig. 17.8, we can see that the keypoints and the correspondences are estimated at the intersection of the shadow and the road surface. As the shadow is moving with the car, these correspondences should be considered outliers. Therefore, it can be stated that the use of neural networks or any semantic network is not a complete solution to suppress the effect of the Independent Moving Objects (IMOs) and there should be advanced techniques that should be employed to solve this problem. Therefore, for the KITTI dataset, I didn't use the YOLO network at all.

On the other hand, semantic labeling can be very useful for maritime sequences where the dominant part of the surrounding environment (sea and fellow boats) is dynamic. In chapter 7,

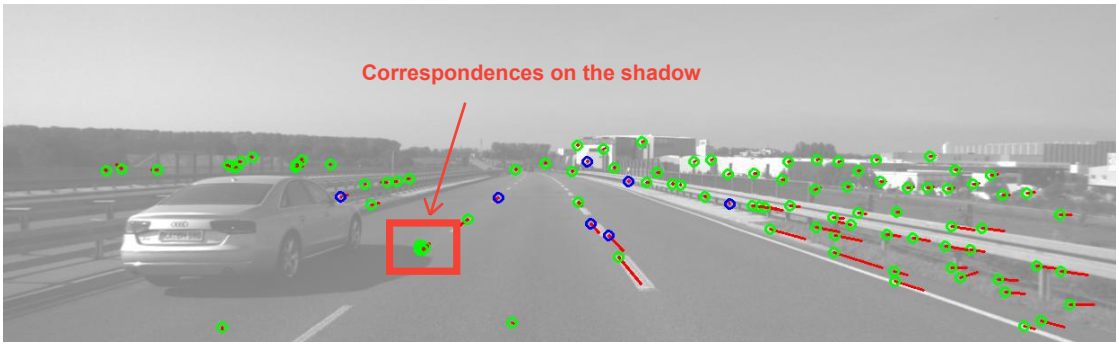
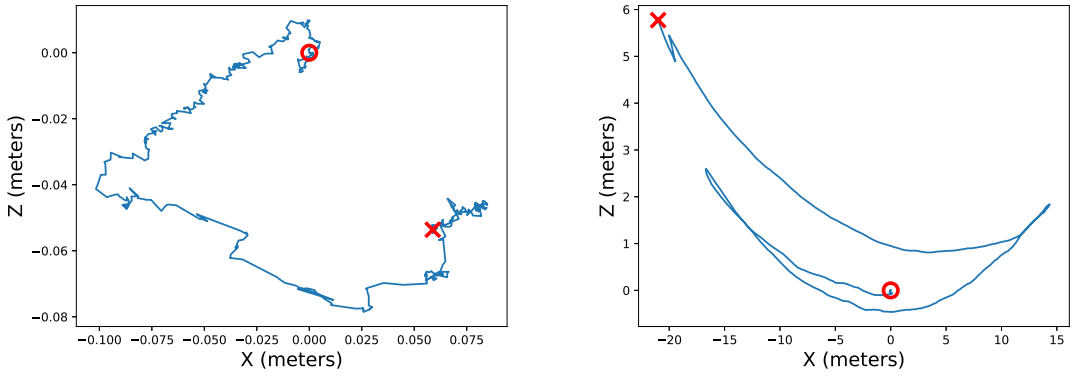


Figure 17.8: The keypoints are extracted at the shadow of the moving vehicle and the correspondences are estimated.



(a) Sub-sequence Z2-FM-EB-SEQ-2-1 (b) Sub-sequence Z2-FM-EB-SEQ-2-2

Figure 17.9: The trajectory for the maritime sequence in the absence of the object masking.

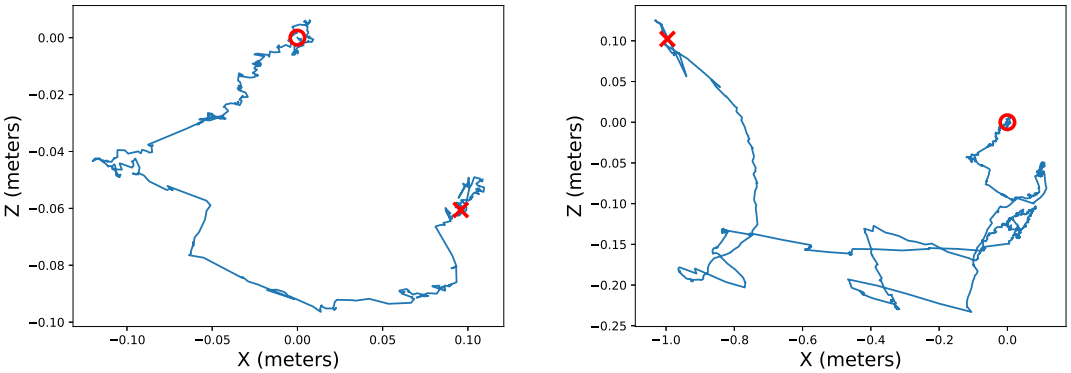
I introduced two semantic networks, in which the Aquanet (see section 7.1.1, p.40) can identify the pixels on the image that lie the sea and the YOLO-v8 can identify the fellow boats. A fused mask generated from these two semantic networks can be very helpful to mask out the IMOs during the keypoints generation.

Figure 17.9 shows the trajectory generated from the seq. Z2-FM-EB-SEQ-2. The sequence is split into two sub-sequences such that the first sub-sequence (Z2-FM-EB-SEQ-2-1) contains the scene in which the ego-vehicle is standing still and the second sub-sequence (Z2-FM-EB-SEQ-2-2) contains the scene in which the ego-vehicle is standing still but observing another boat moving in front of it. In Fig. 17.9a, We can see that the ego-trajectory is drifting around the initial position in the absence of the fellow boat but in Fig. 17.9b, the trajectory is oscillating back and forth as the other fellow boat is moving in a circle in the front of the ego-vehicle and therefore, the ego-motion is strongly influenced.

When the object masking was added to avoid the keypoints lying on the moving objects, the trajectory for the sub-sequence Z2-FM-EB-SEQ-2-1 didn't change much as there was no moving boat in the view but for the sub-sequence Z2-FM-EB-SEQ-2-2, the trajectory got stabled and drifted from the initial position only due to the accumulation of the pose errors over the time.

17.4 RANSAC outlier ratio analysis

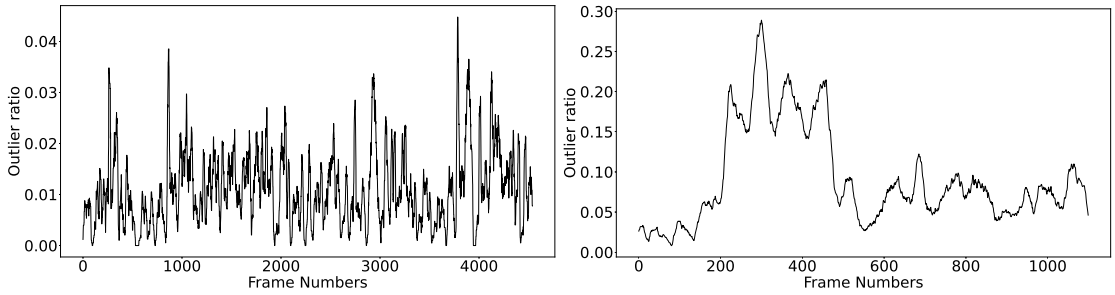
In the current ego-motion estimation system, I have used the RANSAC based PnP method to estimate the pose change from the correspondences (see section 15.2, p.133). The RANSAC is used to make the motion estimation module robust to the outliers in the correspondences and identify the outliers as much as possible such that their effect during the motion estimation can be reduced. It is crucial to check whether the RANSAC actually benefits the algorithm or it is just making it slightly better at the cost of computation. To answer this question, I did a check on the outliers identified by the RANSAC by taking a ratio of the number of outliers to the total number of correspondences estimated at an instant and then taking the moving average (see appendix A.1, p.171) on the outlier ratio for the whole KITTI sequence.



(a) Sub-sequence Z2-FM-EB-SEQ-2-1 (b) Sub-sequence Z2-FM-EB-SEQ-2-2

Figure 17.10: The trajectory for the maritime sequence in the presence of the object masking.

Figure 17.11 shows the outliers ratio test (without using the pyramidal approach) for KITTI seq. 00, and seq. 01 which are the most critical sequences for the analysis. For sequence 01, we can see that the maximum value for the outlier ratio is 0.04 which is 4% of the total number of correspondences (see Fig. 17.11a) but for seq. 01, it reaches to the 0.3 which is the 30% of the total number of correspondences (see Fig. 17.11b). It implies that there were a lot of outliers in the correspondences for seq. 01 but if the RANSAC was able to identify this many outliers and excluded them during the motion estimation then the reason behind the huge position error (see Fig. 17.4c) could either be the inaccurate correspondences in the inliers set or the RANSAC identified the wrong outliers. The analysis of the motion estimation module based on the outliers ratio and position error is left for future work due to the limited time I had on this project.



(a) KITTI Sequence 00 (b) KITTI Sequence 01

Figure 17.11: The ratio of the outliers to the total number of correspondences

CHAPTER 18

Trajectory results for the KITTI and Maritime dataset

Contents

The objective of the ego-motion estimation sub-system is to estimate the pose change between two consecutive timesteps. The pose change can be helpful to analyze the system but in an actual application, we are mostly interested in knowing the absolute pose of the ego-vehicle. This absolute pose can be either w.r.t the world coordinate system or w.r.t the Camera Coordinate Frame (CCF) at timestep $t = 0$. The absolute pose can be generated from the pose change information recorded over time (see section 2.1.1, p.10).

In chapter 17, I mentioned the cases in which the pyramid level can be used by the Correspondence Estimator (CorrEst) module and concluded that the pyramid levels should be avoided if a good pose change predictor is available and not, then the pyramid levels should be used only for the cases where the sharp moments occur. But in some scenarios, the sharp moments of the ego-vehicle can not be avoided especially for the cars. Therefore, I decided to use one pyramid level for the road sequences and no pyramid levels for the maritime sequences. Also, I used object masking for the maritime sequences only.

18.1 Trajectories for the maritime sequences

The trajectories have been plotted for the maritime sequences but due to the unavailability of the ground truth, they can not be trusted completely, however, the shape of the generated trajectory looked similar to the motion of the ego-vehicle in the recorded scenarios. Figure 18.1 shows the trajectory obtained for maritime sequence Z2-FM-EB-SEQ-1. In this sequence, the Milliampere-2 moved straight and then took a right turn to dock to the docking station which is also shown

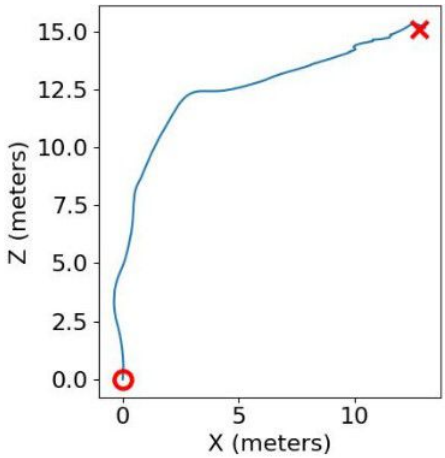


Figure 18.1: The trajectory generated for the maritime sequence Z2-FM-EB-SEQ-1.

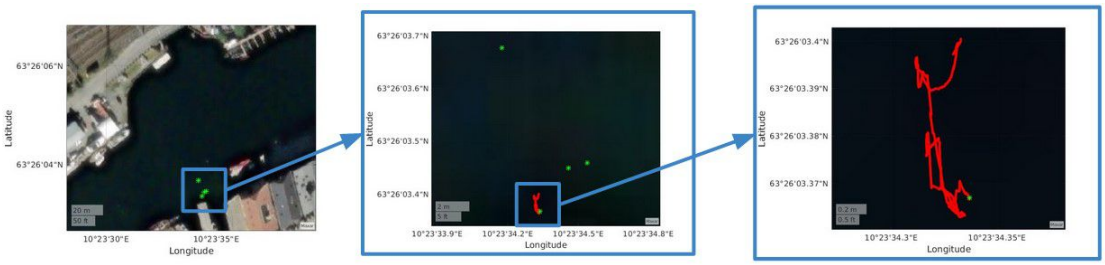


Figure 18.2: The trajectory generated for the maritime sequence Z2-FM-EB-SEQ-2. The green dots are the GPS data and the red trajectory is the estimated trajectory.

by the trajectory however, due to the unavailability of any ground truth, the results can not be verified.

Similarly, figure 18.2 shows the trajectory generated for the maritime sequence Z2-FM-EB-SEQ-2. I managed to obtain a few GPS readings for this sequence from the Garmin GPS device and plotted them on the satellite image using the *geoplot* function from MATLAB. The trajectory obtained from the ego-motion estimation system was in Cartesian coordinates, therefore, it was first transformed into the *North-East-Down (NED)* coordinates and then plotted alongside the GPS data. It should be noted that the ego-vehicle in this sequence was standing still. From the figure, it can be stated that the trajectory is more stable than the GPS data.

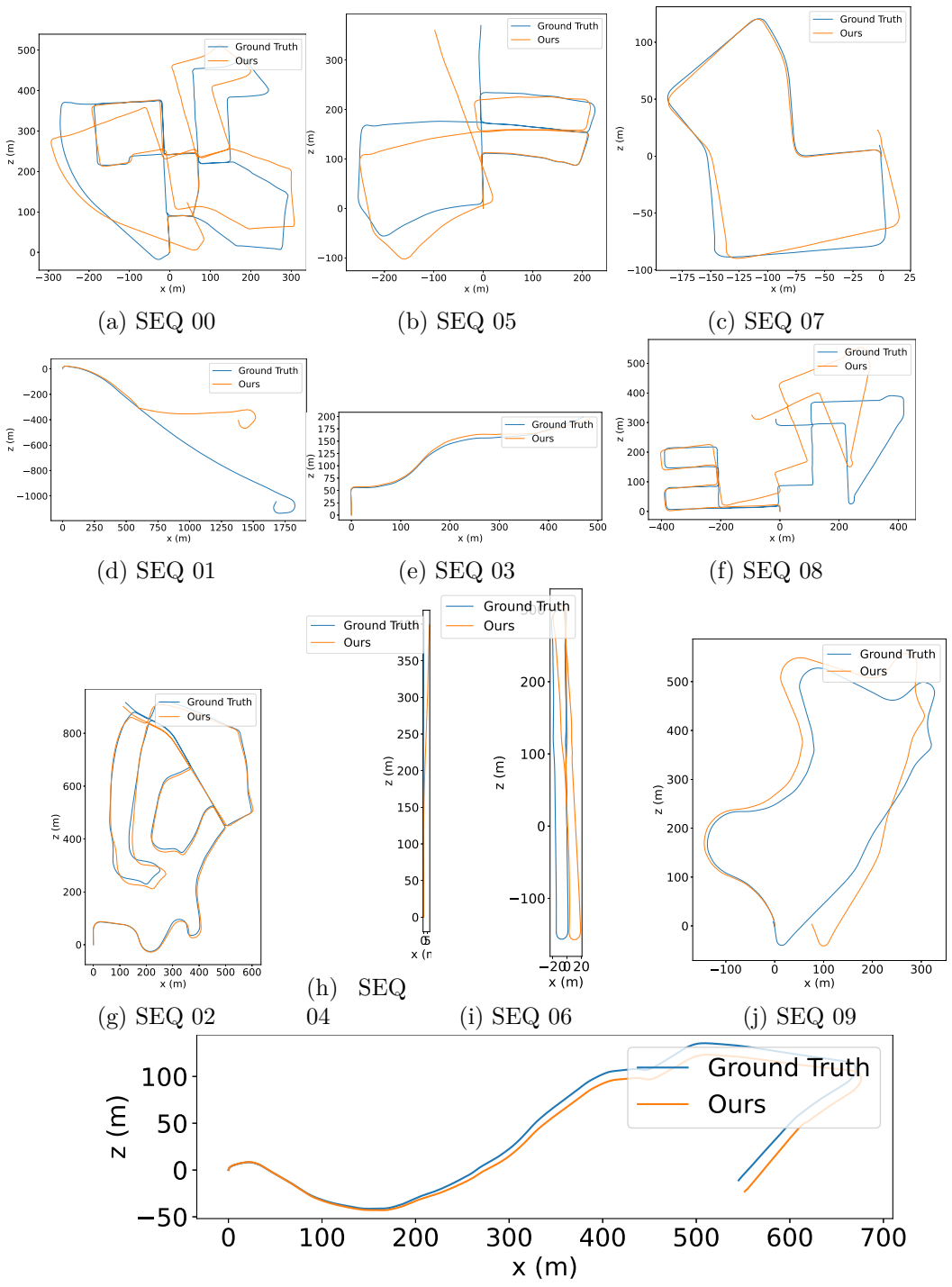
18.2 Trajectories for the KITTI sequences

In this section, I computed the trajectory for the KITTI and compared them against the sequences with the ground truth i.e. the sequences from 00 to 10. To generate the trajectory, I used an open-source toolbox to evaluate the trajectories. The name of the toolbox is *KITTI Odometry Evaluation Toolbox* [1]. Figure 18.3 shows the trajectory estimated using the proposed ego-motion estimation system. In all the estimated trajectories, we can notice that the estimated trajectory

[1]The link to the source code is here

started to diverge from the ground truth sooner or later. In some sequences such as seq. 00, 05, and 08, this divergence is huge, and in others such as seq. 02, 03, 10, it is little.

A final modification that I decided to implement after looking at the trajectories is to use the top k correspondences for the ego-pose estimation. These correspondences have the least residual error as compared to other correspondences. Figure 18.4 shows the new trajectories after using the top k correspondences. The trajectories were improved for most of the sequences. For example, the drift of the estimated trajectory from the ground truth is reduced for seq 00 (see Fig. 18.4a), and the huge position error in seq 01 is also reduced (see Fig. 18.4d).



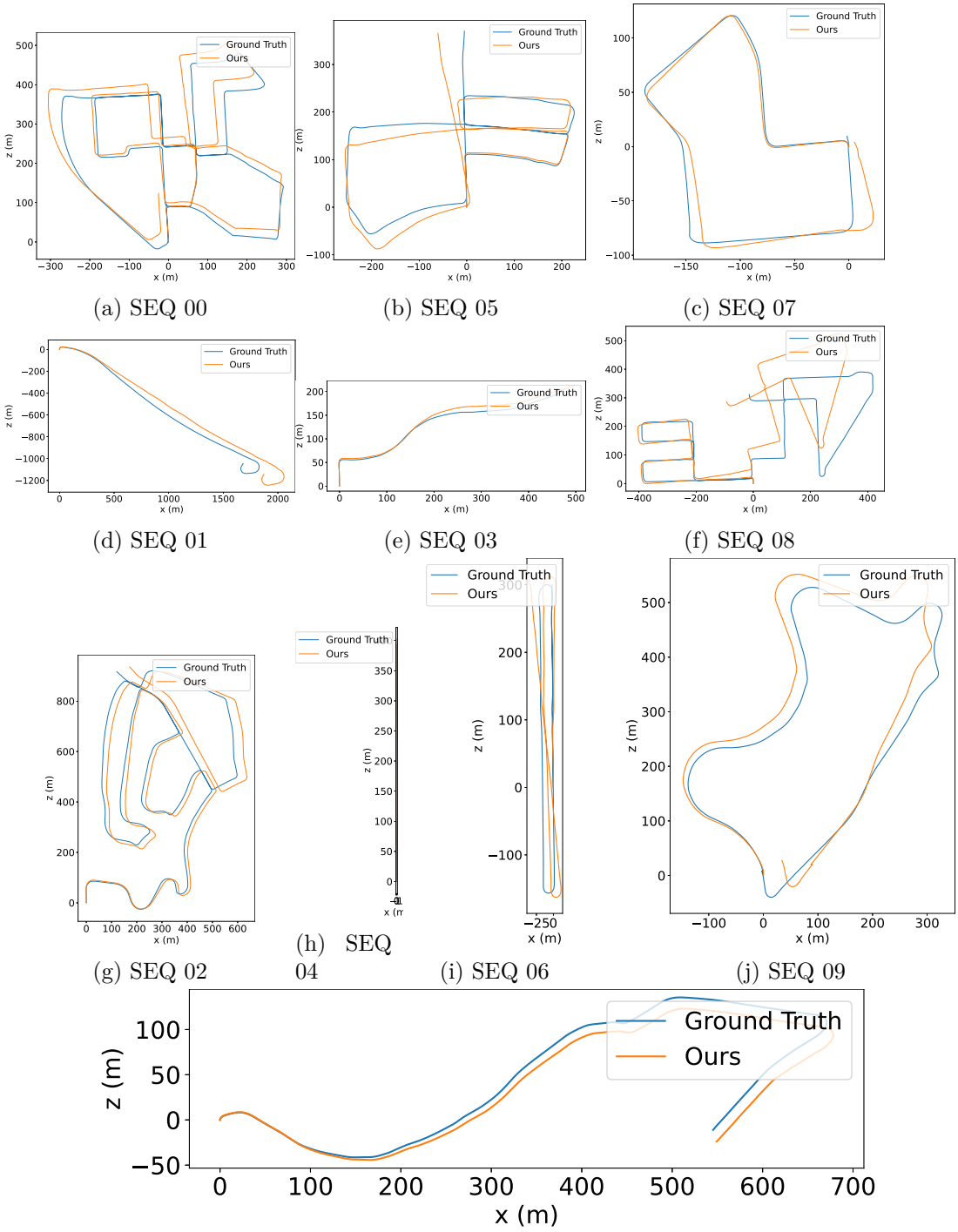
(a) SEQ 00 (b) SEQ 05 (c) SEQ 07

(d) SEQ 01 (e) SEQ 03 (f) SEQ 08

(g) SEQ 02 (h) SEQ 04 (i) SEQ 06 (j) SEQ 09

(k) SEQ 10

Figure 18.3: KITTI Trajectories for different sequences



(a) SEQ 00 (b) SEQ 05 (c) SEQ 07

(d) SEQ 01 (e) SEQ 03 (f) SEQ 08

(g) SEQ 02 (h) SEQ 04 (i) SEQ 06 (j) SEQ 09

(k) SEQ 10

Figure 18.4: KITTI Trajectories for different sequences using the top k correspondences

Part VI

Conclusion and Future work

CHAPTER 19

Conclusion

Ferry-SLAM system is a perception system for maritime vehicles that can represent the surrounding environment using single stereo image pairs and also includes a Visual Odometry (VO) sub-system. In the former, I explored and proposed different strategies to perceive the environment. I used Aquanet and YOLO-v8 semantic segmentation networks to label the seawater and the surrounding obstacles such as a boat, etc. in the image respectively. The YOLO-v8 gave astonishing results while segmenting the fellow boats but it also gave a lot of false positives which made the system more cautious unnecessarily. On the other side, the Aquanet network was found to be good for the proof of concept only as it can not be used in a real application with much reliability. The limitation of the Aquanet provided a motivation to explore the geometry-based methods to segment the water-plane. and therefore, I proposed a novel RANSAC based 3D plane fitting algorithm that can transfer the knowledge of the water plane to the next timestep for robust water surface segmentation. The algorithm can track the water plane with high accuracy even when the ego-vehicle approaches the dock which was not possible with the Aquanet network. The results from the 3D plane fitting algorithm together with the semantic segmentation networks have been fused to create a complete understanding of the environment. This knowledge of the scene can be used to identify the navigable area and the potential objects that should be avoided. I also attempted to follow the Stixels-based approach to represent the scene and proposed a pixel classifier that can distinguish between the upright and the horizontal surfaces in the image using the disparity data only. This type of representation is simpler, computer-efficient, and can be used to identify the navigable areas but the meaning of what it perceives can not be extracted using this representation alone. I used the Bird's Eye View (BEV) for the scene representation which is very effective and can visualize the 3D scene in 2D without losing the depth and the perception information. Such representation can be very helpful in identifying the boundaries of objects such as boats, docks, etc., to avoid collision.

I also developed a frame-to-frame-based ego-motion estimation sub-system that uses the keypoints and their correspondences to estimate the ego-motion. Such keypoints-based systems are very sensitive to the outliers in the correspondences, therefore, the outliers have been identified and rejected at multiple levels to improve the accuracy of the pose estimates. The proposed

Keypoints Detector (KptDet) generates the keypoints in an image using a masking-based strategy that gives full control over the keypoints' generation process to the user. These masks can highlight the region of interest for the keypoint generation without worrying about the distribution and the number of the keypoints in the image and when it is combined with the semantic labels, the perception can be added during the keypoints generation. The perception enabled the KptDet module to avoid generating the keypoints on the sea and the boat because of their dynamic nature to improve the motion estimates. Also, with the help of the depth mask, the keypoints have been generated only for those pixels that have valid depth instead of rejecting them at later stages in the pipeline.

The principle behind the estimation of the rotational angles from the faraway region has been thoroughly studied in this project. I derived the mathematical expression behind this principle and proposed three different novel algorithms based on it. The rotational angles reported in [2] are regarded as the benchmark to compare the accuracy and performance of the proposed methodologies. The first method was to estimate the yaw angle using the phase shift of the distance-weighted horizontal profile of images. This algorithm didn't use much pre-filtering of the image profiles and the post-filtering of the phase change but still, the yaw estimates from this algorithm were close to the ground truth and deviated only during the sharp turns. The second method was also phase correlation (PhC) based and used to estimate the roll and pitch rotational angles along with the yaw angle. The results of this algorithm for the roll and pitch angle were more accurate than the results reported in [2] but for the yaw angle, it didn't perform well comparatively. The third algorithm used the keypoints and their correspondences that lie in the faraway regions and then estimated the rotational angles. The yaw angles estimated from this proposed methodology surpassed the first and second algorithms and the approach mentioned in [2].

For the Correspondence Estimator (CorrEst) module, I implemented a predictive keypoint matching algorithm that uses the prediction of the pose change to predict the location of the keypoints in the next image and then later the correspondences have been optimized using the Lukas-Kanade (LK) differential matcher. The combination of the prediction followed by the optimization reduced the chances of the occurrence of outliers in the correspondences. I also proved experimentally that the pyramidal approach used by the LK matcher can be beneficial only during the sharp turns and should be avoided otherwise but in the absence of a good pose change predictor, the pyramidal approach can compensate for the over and underestimates of the predicted pose change. Finally, the pose change has been estimated using the RANSAC based Perspective-n-Point (PnP) method from the set of the correspondences.

I tested the ego-motion estimation pipeline on the KITTI dataset and the dataset I recorded using the ZED camera in Trondheim, Norway, and compared the generated trajectory with the ground truth from the KITTI dataset. The smooth deviations of the estimated trajectory from the ground truth due to the absence of the loop closure and the accumulation of errors in the estimated pose have been observed in the KITTI dataset. It implies that the ego-motion estimation subsystem managed to address different driving scenarios without running into a failure state. In the recorded sequences, the GPS data was available only for the sequence Z2-FM-EB-SEQ-2 (the ego-vehicle is stationary) and it is being used the compare the estimated trajectory. It was found that the pose of the ego-vehicle was drifting near the initial position even in the presence of other moving boats. Also, the recorded GPS data was more spread around the initial position of the ferry than the estimated trajectory. It was possible only due to semantic masks provided by the YOLO-v8 and the Aquanet during the keypoint generation.

CHAPTER 20

Recommendation of the future work

The Ferry-SLAM project is a long-term project that is under development for the localization, navigation, and situation awareness for maritime vehicles. I started the project and explored different techniques and methods that can be implemented in the maritime context, however, due to the limited time period of the thesis, there were a lot of topics that either remained untouched or didn't end up with a conclusion. In this chapter, I will highlight a few topics that will be the focus in the future of the Ferry-SLAM project.

- **Keyframing:** The keyframing concept is more helpful in the maritime context. When the ferry operates in an open sea and observed only the sea and a distant landmark, the estimation of the change of the position between two consecutive frames is very challenging because the effect of the translation vanishes when the landmark goes to infinity. Therefore, unless there is a huge change in the position of the ferry the system may think that it is only rotating at one position over time but with the keyframing strategy the system can neglect the intermediate frames for a period of time for the estimation of the translation and can estimate the translation only for the keyframes.
- **Improving the rotation estimation using the distance-weighted horizontal profiles:** I proposed an algorithm that can estimate the yaw change between two frames using the phase correlation method: The algorithm needs to be improved such that the yaw estimates can be improved.
- **Keypoint trajectory and Bundle adjustment:** The ego-motion estimation module reuses the old keypoints but doesn't store the trajectory of the keypoints for a number of frames. It can improve the pose estimation and can stable the correspondence estimation. In the future, a suitable datatype should be used to record the trajectory of the keypoints and use them to further optimize the pose estimation using the *Bundle Adjustment*.
- **Improving pose change pre-estimation**: In the current module, I have used only the prediction of the pose change for the correspondence estimation. The predictor can be replaced with some advanced techniques and also the rotation pre-estimates based on

the phase correlation can also be used. I proposed two methods to compute the rotation from far-away areas using phase correlation but I didn't use them in the pose-change pre-estimation due to the limited time. It can be done in the future.

- **Object tracking:** I used YOLO-v8 to identify the fellow boats for the scene representation and to mask them during the keypoints generation but this can be extended to the object tracking in which the objects can be tracked over time to avoid the collision in the future.
- **Stixels-based scene representation:** I implemented the first step towards the stixels-based representation in which I represented the scene in upright and horizontal structures. It was a proof of concept for the future in which the scene can be represented using rectangular strips called stixels.
- **Implementation of the stereo keypoint matching:** The stereo keypoint matching algorithm fully utilizes the stereo camera and can prevent the outliers in the correspondences. I gave a theoretical overview of the same in this project but in the future, the practical aspects should be explored.

Bibliography

[1] Hernán Badino, Uwe Franke, and David Pfeiffer. The stixel world-a compact medium level representation of the 3d-world. In *Pattern Recognition: 31st DAGM Symposium, Jena, Germany, September 9-11, 2009. Proceedings 31*, pages 51–60. Springer, 2009.

[2] Marc Barnada, Christian Conrad, Henry Bradler, Matthias Ochs, and Rudolf Mester. Estimation of automotive pitch, yaw, and roll using enhanced phase correlation on multiple far-field windows. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 481–486. IEEE, 2015.

[3] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer vision and image understanding*, 110(3):346–359, 2008.

[4] Steven S. Beauchemin and John L. Barron. The computation of optical flow. *ACM computing surveys (CSUR)*, 27(3):433–466, 1995.

[5] Jean-Yves Bouguet et al. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel corporation*, 5(1-10):4, 2001.

[6] Henry Bradler, Birthe Anne Wiegand, and Rudolf Mester. The statistics of driving sequences – and what we can learn from them. In *2015 IEEE International Conference on Computer Vision Workshop (ICCVW)*, pages 106–114, 2015.

[7] Edmund F. Brekke. Optical flow for tracking maritime objects, Apr 2022.

[8] Edmund F Brekke, Egil Eide, Bjørn-Olav H Eriksen, Erik F Wilthil, Morten Breivik, Even Skjellaug, Øystein K Helgesen, Anastasios M Lekkas, Andreas B Martinsen, Emil H Thyri, et al. milliampere: An autonomous ferry prototype. In *Journal of Physics: Conference Series*, volume 2311, page 012029. IOP Publishing, 2022.

[9] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In *Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV 11*, pages 778–792. Springer, 2010.

[10] Marius Cordts, Timo Rehfeld, Lukas Schneider, David Pfeiffer, Markus Enzweiler, Stefan Roth, Marc Pollefeys, and Uwe Franke. The stixel world: A medium-level representation of traffic scenes. *Image and Vision Computing*, 68:40–52, 2017.

[11] Binge Cui, Wei Jing, Ling Huang, Zhongrui Li, and Yan Lu. Sanet: A sea–land segmentation network via adaptive multiscale feature learning. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 14:116–126, 2021.

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[13] David W Eggert, Adele Lorusso, and Robert B Fisher. Estimating 3-d rigid body transformations: a comparison of four major algorithms. *Machine vision and applications*, 9(5-6):272–290, 1997.

[14] Seyed Mohammad Hassan Erfani, Zhenyao Wu, Xinyi Wu, Song Wang, and Erfan Goharian. Atlantis: A benchmark for semantic segmentation of waterbody images. *Environmental Modelling & Software*, page 105333, 2022.

[15] Nolang Fanani. *Predictive Monocular Odometry Using Propagation-Based Tracking*. PhD thesis, Goethe University Frankfurt, Frankfurt am Main, Germany, 2018.

[16] Nolang Fanani. *Predictive monocular odometry using propagation-based tracking*. doctoralthesis, Universitätsbibliothek Johann Christian Senckenberg, 2018.

[17] Friedrich Fraundorfer and Davide Scaramuzza. Visual odometry: Part i: The first 30 years and fundamentals. *IEEE Robotics and Automation Magazine*, 18(4):80–92, 2011.

[18] Friedrich Fraundorfer and Davide Scaramuzza. Visual odometry: Part ii: Matching, robustness, optimization, and applications. *IEEE Robotics & Automation Magazine*, 19(2):78–90, 2012.

[19] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[20] Simon Godsill. Particle filtering: the first 25 years and beyond. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7760–7764, 2019.

[21] Chris Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[22] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[23] Øystein Kaarstad Helgesen, Emil H Thyri, Edmund Brekke, Annette Stahl, and Morten Breivik. Experimental validation of camera-based maritime collision avoidance for autonomous urban passenger ferries. 2023.

[24] H. Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 807–814 vol. 2, 2005.

[25] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):328–341, 2008.

[26] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. 1960.

[27] Hoa Dang Khanh, Van Son Nguyen, Anh Do, and Nguyen Tien Dzung. An effective randomized hough transform method to extract ground plane from kinect point cloud. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1053–1058, 2019.

[28] Rifat Kurban, Florenc Skuka, and Hakki Bozpolat. Plane segmentation of kinect point clouds using ransac. In *7th International Conference on Information Technology, ICIT, Amman, Jordan*, pages 545–551, 2015.

[29] Guichi Liu, Enqing Chen, Lin Qi, Yun Tie, and Deyin Liu. A sea-land segmentation algorithm based on sea surface analysis. In Enqing Chen, Yihong Gong, and Yun Tie, editors, *Advances in Multimedia Information Processing - PCM 2016*, pages 479–486, Cham, 2016. Springer International Publishing.

[30] Hao Liu, Dan Dan Huang, and Zhen Ye Geng. Visual odometry algorithm based on deep learning. In *2021 6th International Conference on Image, Vision and Computing (ICIVC)*, pages 322–327, 2021.

[31] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60:91–110, 2004.

[32] Bruce D Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI'81: 7th international joint conference on Artificial intelligence*, volume 2, pages 674–679, 1981.

[33] David Nistér. An efficient solution to the five-point relative pose problem. *IEEE transactions on pattern analysis and machine intelligence*, 26(6):756–770, 2004.

[34] Matthias Ochs, Henry Bradler, and Rudolf Mester. Enhanced phase correlation for reliable and robust estimation of multiple motion distributions. In Thomas Bräunl, Brendan McCane, Mariano Rivera, and Xinguo Yu, editors, *Image and Video Technology*, pages 368–379, Cham, 2016. Springer International Publishing.

[35] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[36] Jhony Kaesemodel Pontes, James Hays, and Simon Lucey. Scene flow from point clouds with or without learning. In *2020 International Conference on 3D Vision (3DV)*, pages 261–270, 2020.

[37] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[38] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Computer Vision–ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I 9*, pages 430–443. Springer, 2006.

[39] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. Ieee, 2011.

[40] Willem P. Sanberg, Gijs Dubbelman, and Peter H. N. de With. Free-space detection with self-supervised and online trained fully convolutional networks, 2017.

[41] Peter H Schönemann. A generalized solution of the orthogonal procrustes problem. *Psychometrika*, 31(1):1–10, 1966.

[42] Pourya Shamsolmoali, Masoumeh Zareapoor, Ruili Wang, Huiyu Zhou, and Jie Yang. A novel deep structure u-net for sea-land segmentation in remote sensing images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 12(9):3219–3232, 2019.

[43] Jianbo Shi et al. Good features to track. In *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, pages 593–600. IEEE, 1994.

[44] Peter Sturm. Pinhole camera model, 2014.

[45] Juan Terven and Diana Cordova-Esparza. A comprehensive review of yolo: From yolov1 and beyond, 2023.

[46] Emanuele Trucco and Alessandro Verri. *Introductory techniques for 3-D computer vision*, volume 201. Prentice Hall Englewood Cliffs, 1998.

[47] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.

Appendix A

Mathematical Appendix

Contents

A.1 Moving average

I analyzed the Ferry-SLAM system by plotting different statistics for different KITTI sequences. These include the number of keypoints generated in each frame, the number of valid correspondences, the yaw error per frame, etc. If I had plotted the raw data, It could have been very difficult to analyze and find the critical sub-sequences in which the system was underperforming. Therefore, I smoothed the data by convolving it with a kernel of size w[1]. In the moving average, the function slides a window of length w over the input data (number of generated keypoints per frame for say) such that the oscillations can be minimized.

A.2 Image contrasting and visualization of the depth images

Throughout the report, I have used multiple images from the KITTI dataset and the recorded dataset and overlayed the graphics over them for the explanation of the modules of the system.

[1] In the convolution, I used the Numpy function *convolve* with the *valid* mode. In this mode, the moving average is taken only for those which completely overlap with the lowpass filter.

For example, I plotted the detected keypoints on the images. Such graphics are not clearly visible if they are overlaid over the raw image from the dataset, therefore, I first controlled the contrast on the input image only for the presentation of the data in this report. Let (x, y) be the coordinates of the pixel, $I(x, y)$ be the intensity of the pixel (x, y) with a range $[0, max_I]$, and $I'(x, y)$ be the intensity of the same pixel after controlling the contrast, then the contrast can be reduced by reducing the range of the intensity of the pixel.

$$I'(x, y) = \frac{max_I}{2} + \frac{I(x, y)}{2} \tag{A.1}$$

Similarly, the depth maps are 16 bits floating number arrays and they can not be visualized directly by an 8 bits integer array which is the most common data format for the images. In the literature, the authors always use false coloring for the visualization of the depth images. In such color mappings, the blue color can represent the areas that are close to the camera and the red for the distant areas. I tried to follow such color mapping for the visualization of my depth images. Let D be a depth image with 16 bits floating number, then the depth image D can have invalid values in the data. These invalid values can come from the occluded pixels or the regions in the image that are very far or close to the camera. These invalid values need to be replaced with some integer before the visualization. I replaced them with zero. Let (x, y) be the pixel coordinates, $D(x, y)$ be the depth of the pixel (x, y), and $D'(x, y)$ be the filtered depth, then the invalid values can be replaced with zero to filter the raw depth image D.

$$D'(x, y) = \begin{cases} D(x, y), \text{if } D(x, y) = valid \\ 0, \text{otherwise} \end{cases} \tag{A.2}$$

After the filtering, the depth image D' is still 16 bits floating array. Therefore, it should be scaled down to 8 bits integer array. I used the *convertAbsScale*[2] function from the OpenCV library. This function performs three operations sequentially: scaling, taking an absolute value, and conversion to an unsigned 8-bit type. Let α and β be the scaling factors then they can be calculated from the range of the filtered depth image D'.

$$\begin{aligned} \alpha &= \frac{max_I}{\max(D') - \min(D')} \\ \beta &= 0 \end{aligned} \tag{A.3}$$

Let D'' be the scaled depth image, then the function *convertAbsScale* applies the absolute scaling to the filtered depth image D' as below.

$$D'' = saturate\_cast < uchar > (|D'\alpha + \beta|) \tag{A.4}$$

Finally, the *Turbo* color mapping has been applied to the scaled 8 bits depth image D'' using the *applyColorMap* function from the OpenCV.

A.3 Sigmoid curve

In the Rotation estimation from faraway areas using the distance-weighted gray value profiles method (see section 14.9, p.118), I wanted to weight the grayscale patches with their corresponding depth patches such that the pixels that are far away are given more importance. A hard

[2]The link to the function is here.

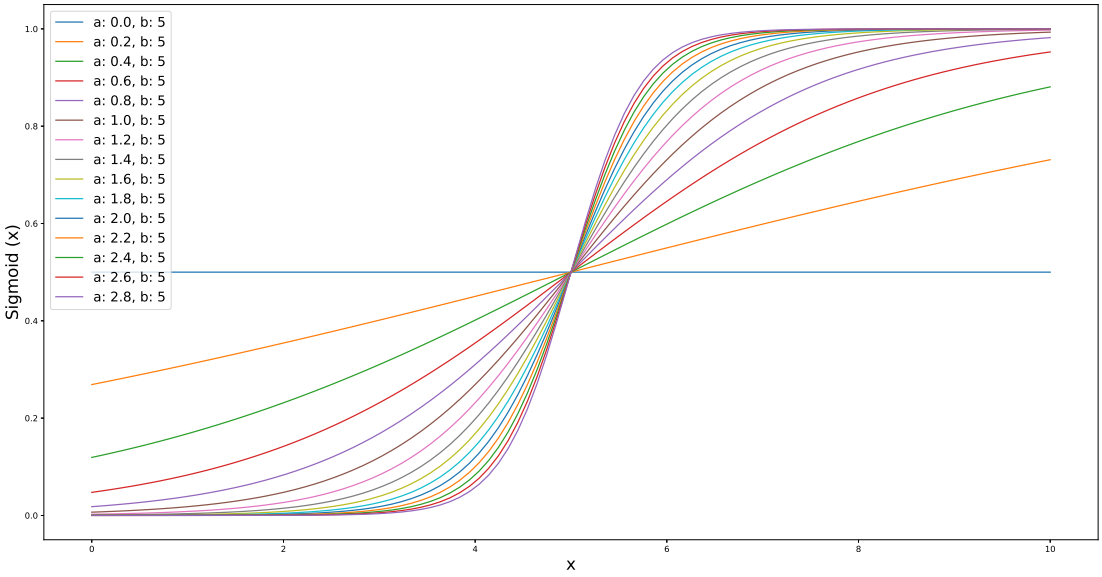


Figure A.1: Sigmoid curve with different slope and fixed horizontal displacement

threshold on the depth patch can be used to prepare a binary weighted mask of zeros and ones where the ones are given to the pixels that have a depth higher than the threshold, however, this totally ignores the close-range pixels which are crucial during the turns in the urban scenarios where the close range pixels are dominating. Therefore, I passed the depth patch to a sigmoid function to smoothly assign the weights depending on the depth.

Let a and b be the sigmoid parameters, and x be the input to the function. With this, the sigmoid of the input x can be computed.

$$\sigma(x) = \frac{1}{1 + \exp -a(x - b)} \tag{A.5}$$

A.4 Conversion of the rotation matrix to Euler angles

If a rotation matrix $\mathbf{R}$ of size 3×3 is given with a determinant $+1$, then it can be decomposed into the Euler angles as follows [3].

[3]The tutorial to the conversion of rotational angles is given here

Algorithm 23 Rotation matrix to Euler angles

```
#Define EPS ← 0.000001
procedure rotation2euler(R)
    sy = √(R[0,0]² + R[1,0]²)
    if sy > EPS then
        rot_x = atan2(R[2,1], R[2,2])
        rot_y = atan2(−R[2,0], sy)
        rot_z = atan2(R[1,0], R[0,0])
    else
        rot_x = atan2(−R[1,2], R[1,1])
        rot_y = atan2(−R[2,0], sy)
        rot_z = 0
    return rot_x, rot_y, rot_z
```

A.5 Conversion of the Euler angles to rotation matrix

Mathematically, the rotation can be expressed as a 3×3 orthogonal matrix $\mathbf{R}$. If the rotation about the $x - axis$, $y - axis$, and $z - axis$ are denoted θ, ψ, and ϕ, then the rotation matrices for each axis can be computed.

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{A.6}$$

$$\mathbf{R}_y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix} \tag{A.7}$$

$$\mathbf{R}_z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.8}$$

The rotation matrices $\mathbf{R}_x(\theta)$m $\mathbf{R}_y(\psi)$, and $\mathbf{R}_z(\psi)$ can be multiplied together to form the rotation matrix $\mathbf{R}$. It should be noted that the order of the multiplication matters.

$$\begin{aligned} R(\phi,\psi,\theta) &= euler2rotation(\phi,\psi,\theta) \\ &= R_z(\phi)R_y(\psi)R_x(\theta) \\ &= \begin{bmatrix} \cos\phi\cos\psi & \cos\phi\sin\psi\sin\theta - \cos\theta\sin\phi & \sin\phi\sin\theta + \cos\phi\cos\theta\sin\psi \\ \cos\psi\sin\phi & \cos\phi\cos\theta + \sin\phi\sin\psi\sin\theta & \cos\theta\sin\phi\sin\psi - \cos\phi\sin\theta \\ -\sin\psi & \cos\psi\sin\theta & \cos\psi\cos\theta \end{bmatrix} \end{aligned} \tag{A.9}$$

Appendix B

Multi plane identification

It is always interesting to identify all the planes in the scene for a simple and intuitive scene representation, so I decided to extend the Adaptive Plane Segmentation using RANSAC (APSR) (see section 7.2.8, p.47) to identify more planes from the scene. After identifying the water plane, a recursive loop was created to keep identifying the planes from the scene until the number of points to be fitted dropped below a threshold. The steps followed by the extended algorithm are as follows.

1. *Water Plane Segmentation:* The water plane is extracted from the masked point cloud using Adaptive Plane Segmentation using RANSAC (APSR) approach. This results in a mask of inliers for the points that lie on the water plane.
2. *Cropping:* The raw point cloud is cropped using the inliers mask generated in the previous step such that the cropped point cloud has no point on the water plane. Let us say it an outlier point cloud.
3. *Recursive plane identification:*
    a) Number of points in the outlier point cloud is checked. If they are below a threshold, then no new plane can be extracted and the algorithm ends.
    b) In the outlier point cloud, a plane is fitted using Open3D's plane segmentation algorithm. Apart from the plane model, it also returns the new outlier point cloud that can be used for the next iterations.
    c) The inliers are found using 7.2.5.

The above approach is very simple and in theory, it should find all the planes recursively. From the figure B.1, it is clear that the algorithm didn't perform as excepted. It managed to find only one other plane which was the dock even after a lot of iterations. The points that lie on the segmented dock plane seem to overlap with the water plane because the estimated dock plane is slanted and not completely parallel to the water plane. It implies either that the dock

Table B.1: Cropping limits for the point cloud

Axis	Range (in meters)	
	Minimum	Maximum
x	-10	10
y	-5	5
z	0	20

plane model can not be trusted or the dock is actually slanted with respect to the water plane. The major flaws of this approach are the following.

- When the ferry is far from the dock, the fitting of planes on the structure became very hard because of the high variance in the disparity. The disparity is accurate only up to short distances.
- Determining a fixed distance threshold is not enough to identify different kinds of planes. Thresholds should be adaptive and vary according to the distance to the structures.
- The number of iterations performed by the RANSAC has to increase to make it identify the plane among the large set of outliers but the computational time will be increased significantly.

To solve the first two drawbacks of the algorithm, I limited the search of planes within a close range only. It means, instead of taking the whole outlier point cloud, the algorithm crops the outlier point cloud such that the points that lie outside the given range (refer table B.1) are excluded. Defining a close-range outlier point cloud reduces the occurrences of false disparity measurements in the plane segmentation pipeline. The threshold can be limited to some value as well for all the planes. After this change in the algorithm, it is not able to identify the other planes unless the ego-vehicle is close to the dock. Figure B.2, shows the results of the multi-plane segmentation approach when the ferry is close to the dock. From the results, it is very clear that the multiple planes can be found recursively but only in certain situations which are; good disparity estimates; and a large number of RANSAC iterations.

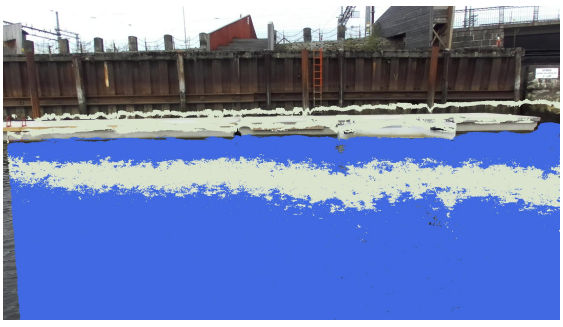


Figure B.1: Multi plane segmentation. Navy blue shows the water plane and the cream color shows the second plane.

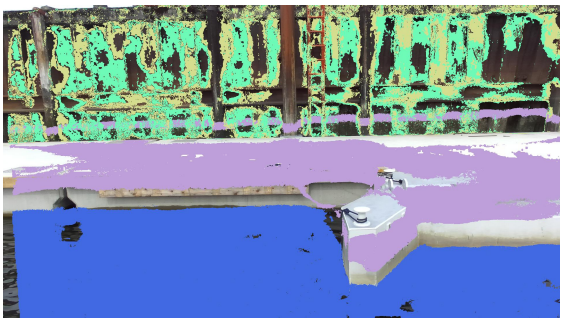


Figure B.2: Multi plane segmentation after the cropping of the outlier point cloud. The navy blue color represents the water plane and the other 3 colors represent the other identified planes.

APPENDIX C

Design and Test Principles

Contents

The theoretical aspect and results of the work are very important to understand the fundamentals, methods, and algorithms used in this project but at the same time, the practical aspects should not be neglected as well. In this chapter, I will briefly talk about the design and testing principles I followed to build the Ferry-SLAM system.

C.1 Design principles

The main design principles that I considered while developing the system are the following.

- **Modularity:** The Ferry-SLAM system consists of multiple modules in which each of the modules has unique functionality. It is very common to test new methods and techniques during the development of the system but if the system is not modular then the chances of the failure of the system increase or not but the time of the development, debugging, and testing increase for sure. Therefore, I tried to develop the Ferry-SLAM system as modular as I could such that the new methods can be introduced without changing the main architecture of the system as well as the readability of the code can be increased for future work.
- **State awareness:** For any complex system that uses multiple sub-systems and modules such as the Ferry-SLAM system, it is very often that one of its components breaks during the processing of the data or runs into a questionable state that the system was not prepared for. It is impossible to know every state of a system that operates in a very dynamic environment such as the roads, sea, etc. Therefore, the system should be aware of such questionable states and warn the user if it falls into any such situation. I attempted to

follow the same principle and returned the status from every module. The most common statuses were the *SUCCESS* and the *FAILURE*. If any module processes the input successfully, it returns the *SUCCESS*, and if not then it either returns a *FAILURE* status or another specially designed error code. For example, if the Keypoints Detector (KptDet) module can not find any Good Features to Track (GFTT) keypoint in the input image, then it returns the *NO_GFTT_FOUND* status. Similarly, if the number of keypoint-to-keypoint correspondences is not sufficient for the motion estimation, then the module returns *NOT_ENOUGH_CORRESPONDENCES* status. Using these status messages, the super system that is using the Ferry-SLAM system can take the precautionary steps whenever my system runs into such state.

C.2 Testing priciples

The principles or guidelines I followed for the analysis and the testing are the following.

- **Identification of edge cases:** A most favored procedure in algorithm development is to run the algorithm on some test dataset and identify the edge cases and failure cases. These cases can be distinguished from the rest by the huge deviations from the expected output and these cases should be handled carefully. I followed the same approach and identified the critical sub-sequences from the KITTI dataset in which my system was not performing well. I focused on different statistics such as the pose error or the number of keypoints generated (see chapter 17, p.143) for the analysis of the module and put the critical sub-sequence into the control file of the sequencer (see section 6.1.3, p.37) to analyze it again.

- **Regression tests:** Whenever I identified an edge case and analyzed it by looking at different statistics, I fine-tuned the system parameters such that the edge cases and the failure cases were resolved and then ran the whole system again to check if the modified version still works fine with the data it could process successfully before.

C.2.1 Data collection for testing and analysis

For the ego-motion estimation sub-system, the most important parameters are the pose parameters. The estimated pose needs to be compared with the ground truth to check the accuracy of the sub-system, therefore, along with different parameters, the pose is also recorded for a sequence. In this section, I will discuss how the pose data has been recorded and what other parameters have been recorded.

In chapter 10, we saw that whenever the new stereo data arrives from the camera, the ego-motion estimation sub-system outputs the relative pose of the camera between the current timestep t and the previous timestep $t-1$. Let $\vec{P}_{t-1}$ be a 3D homogeneous point w.r.t the CCF at timestep $t-1$ and $\mathbf{T}_{t-1}^{t}$ be the homogeneous transformation matrix that defines the relative pose of the CCF at the previous timestep $t-1$ w.r.t the CCF at the current timestep t, such that the point $\vec{P}_{t-1}$ can be transformed to the CCF at timestep t using the transformation matrix $\mathbf{T}_{t-1}^{t}$.

$$\vec{P}_t = \mathbf{T}_{t-1}^{t}\vec{P}_{t-1} \tag{C.1}$$

When the algorithm is run over a sequence of images, I will obtain $N-1$ relative poses from the pipeline from a sequence containing N frames. Each relative pose $\mathbf{T}_{t-1}^{t}$ obtained at timestep

t can be decomposed into the rotation matrix ($\mathbf{R}_{t-1}^{t}$) and the translation vector ($\vec{t}_{t-1}^{\ t}$).

$$\mathbf{T}_{t-1}^{t} = \begin{bmatrix} \mathbf{R}_{t-1}^{t} & \vec{t}_{t-1}^{\ t} \\ 0_{1\times 3} & 1_{1\times 1} \end{bmatrix} \tag{C.2}$$

The rotation matrix can be further decomposed into the Euler angles (see algorithm 23, p.174). Let rot_x, rot_y, and rot_z be the rotational angles about the $x-axis$, $y-axis$, and $z-axis$ respectively w.r.t the CCF.

$$rot_x, rot_y, rot_z = rotation2euler(\mathbf{R}_{t-1}^{t}) \tag{C.3}$$

The rot_x, rot_y, and rot_z rotational angles and the translation vector $\vec{t}_{t-1}^{\ t}$ between two consecutive stereo pair is saved along with other miscellaneous data. The miscellaneous data consist of the intermediate information given by the pipeline that can be useful to debug. It includes the execution time, and the system status such as *SUCCESS*, *FAILURE*, etc. The keypoints and the correspondence's statistics have also been recorded during the sequence. All of this information can be useful to create a trajectory from the relative poses as well as to debug the pipeline.

APPENDIX D

ZED Stereo Camera

Contents

ZED is a stereo camera developed by Stereo Labs company. It is specially designed for Computer Vision (CV) application as it comes with a proprietary SDK that allows the user to use the camera in their autonomous applications. It provides multiple features that can be used from the Visual SLAM (V-SLAM) perspective. In this chapter, I will discuss the specifications and features of the ZED camera.

D.1 Camera Specifications

The Stereo Labs manufactures different types of stereo cameras. A brief comparison between different models is given in table D.1 and table D.2. The basic model from the Stereo Labs is called ZED 1 camera (also used in this project) and it can be used with different resolutions such as 2k High Definition (HD), HD 1080, HD 720, and WVGA. The intrinsic properties such as the FOV, focal length, etc. of the camera varies depending upon the resolution used. Table D.3 gives the approximate values of the different intrinsic parameters of the ZED 1 camera. The intrinsic calibration parameters of the ZED 1 camera that has been used in this project are given in table D.4. It should be noted that the calibration of the camera was done by the company itself.

The default settings of the camera allow us to retrieve the rectified images, however, it is also possible to get the raw data from the cameras as well as the grayscale images, depth images, and normal's images (color rendering of the normals).

Table D.1: Operational characteristics of different ZED camera models

Model	Depth Range	Depth Accuracy	Operating Temperature (in Celsius)
ZED	0.5 m to 25 m	$< 2\%$ up to 3m $< 4\%$ up to 15m	0° to +45°
ZED M	0.10 m to 15 m	$< 1.5\%$ up to 3m $< 7\%$ up to 15m	0° to +45°
ZED 2	0.3 m to 20 m	$< 1\%$ up to 3m $< 5\%$ up to 15m	-10° to +45°
ZED 2i 4mm	1.5 m to 35 m	$< 2\%$ up to 10m $< 7\%$ up to 30m	-10° to +45°
ZED X 4mm [1]	1.5 m to 35 m	$< 2\%$ up to 10m $< 7\%$ up to 30m	-20° to +55°

Table D.2: Available sensors and miscellaneous features of different ZED camera models.

Model	IMU	Magnetometer	Barometer	Temperature sensor	IP66-rated Enclosure
ZED	✗	✗	✗	✗	✗
ZED M	✓	✗	✗	✗	✗
ZED 2	✓	✓	✓	✓	✗
ZED 2i 4mm	✓	✓	✓	✓	✓
ZED X 4mm	✓	✗	✗	✓	✓

Table D.3: Approximate values of the intrinsic properties of the ZED camera for different resolutions.

Resolution	Focal length (in pixels)	Pixel size (in mm)	FOV		Available FPS
			Vertical	Horizontal	
HD2K	1400	0.002	47°	76°	15
HD1080	1400	0.002	42°	69°	15, 30
HD720	700	0.004	54°	85°	15, 30, 60
WVGA	350	0.008	56°	87°	15, 30, 60, 90

D.2 Depth Sensing

ZED camera's SDK also provides the 3D data directly without the explicit processing of the rectified images. The maximum recommended depth range of the camera is from 0.4 meters to 40 meters but it can be configured according to the application. The SDK provides different methods to compute the depth map from the stereo data. The available methods are the following.

[1] The moment when this report was written, the model ZED X has not been launched. This new model offers ultra-wide lenses (up to 120 degrees) and can provide a depth map at 120 FPS at 600p resolution. It is also possible to obtain the HD1080 or HD1200 resolution 60 FPS which is not possible in other models.

Table D.4: ZED's left camera calibration parameters at 2k HD resolution.

Parameter name	Parameter Value (in pixels)
f_x	1404.2700
f_y	1404.8199
c_x	1110.9700
c_y	614.7650

- **Neural:** It uses AI techniques to provide accurate and smooth depth maps (see Fig. D.1).
- **Ultra:** It gives the highest depth range with improved depth accuracy.
- **Quality:** It gives smooth surfaces using strong filtering techniques.
- **Performance:** It provides smooth depth maps but lacks some details.

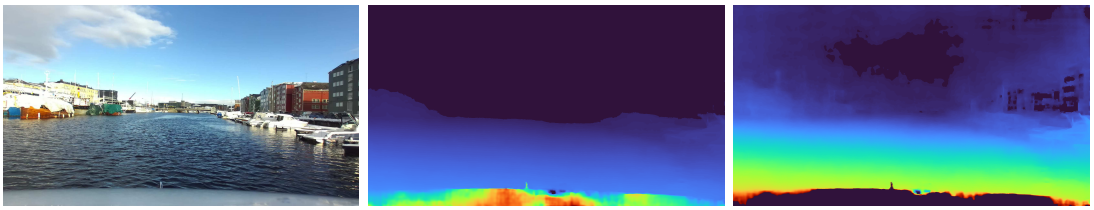
The depth map obtained from these depth modes can have some holes in them. These holes arise from the occlusions and failure of the stereo-matching algorithm. The ZED SDK provides additional settings called FILL (see Fig. D.2b) in contrast to the STANDARD mode (see Fig. D.2a) to fill the holes and occlusions in the depth map and improves edges and temporal stability by adding a filtering stage. From the manufacturer, it is recommended that the FILL mode should be preferred for visualization only.

In the case of the STANDARD mode, the depth image may have some invalid depth data (holes) for some pixels of the image. The values associated with the invalid depth can take the following depending on the situation.

- *NAN*: The depth can not be estimated due to the occlusion.
- - *INFINITY*: The depth can not be estimated because it is too close to the camera.
- *INFINITY*: The depth can not be estimated because it is too far from the camera.

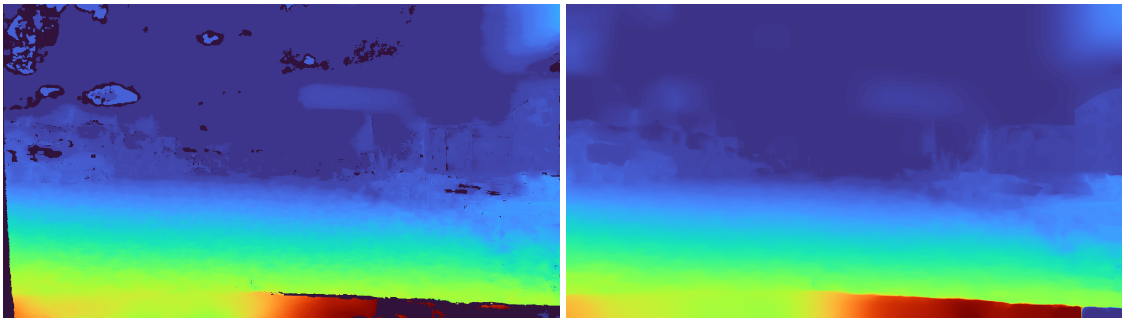
D.3 Depth Map Filtering

The depth map provided by the ZED camera can be filtered according to the confidence of the algorithm in estimating the depth and the texture of the scene.



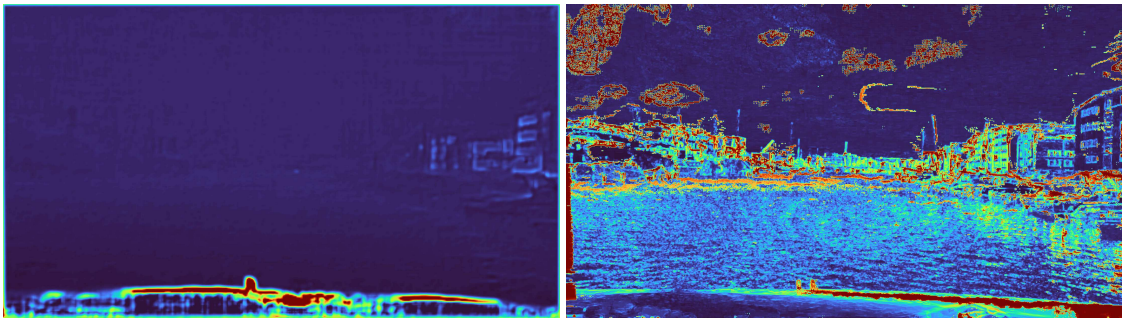
(a) Color image (b) Depth range: 0.4m to 40m (c) Depth range: 10m to 200m

Figure D.1: Images from seq. Z1-FT-MB-SEQ-1. The middle and right figures are the depth images computed using Neural mode with different depth ranges.



(a) Standard mode (b) Fill mode

Figure D.2: Depth images computed using Ultra mode with depth range 10m to 200m.



(a) Neural mode (b) Ultra mode

Figure D.3: Confidence maps for different depth sensing modes computed with a depth range of 10m to 200m with FILL mode.

D.3.1 Confidence Filtering

During the depth map computation, any algorithm tries to assign a depth value to each pixel if a corresponding pixel on the other image of the stereo pair is found. The assignment of depth is never 100% accurate and therefore, a confidence value can also be assigned in parallel to the depth value. This confidence value corresponds to the accuracy of the algorithm in estimating the depth. For example, the algorithm may be less confident to assign the depth at the boundaries between two objects.

The ZED SDK also provides a *confidence map* (see Fig. D.3) along with the depth map. It gives every pixel in the image a value in the range [1,100]. Pixels having a confidence value close to 100 are not to be trusted. The depth map can be filtered using this confidence map explicitly or it can also be done by the SDK itself. A runtime parameter (*confidence_threshold*) can be used to remove all points from the data that have confidence higher than this threshold.

D.3.2 Texture Filtering

In most of the VO or the V-SLAM applications, the part of the scene that has enough texture is utilized for keypoint extraction and matching as the textured region can be easily tracked without ambiguity. Therefore, such algorithms always check the texture of the scene before

going further.

Unlike the confidence map, it is not possible to get the textured map from the ZED SDK, however, it is possible to filter the depth map based on the texture level of the image. A parameter (*texture_confidence_threshold*) can be set between 1 and 100 as a runtime parameter to the SDK. This will remove all the points from the data that have a texture value above this threshold. It is counter-intuitive because the texture value is inversely proportional to the level of texture. For example, the $texture\_confidence\_threshold = 100$ will keep all the regions of the image (uniform and non-uniform) whereas $texture\_confidence\_threshold = 50$ may remove the uniform areas from the depth map. The *texture_confidence_threshold* can take a value in the range [1, 100].

D.4 Advanced Features

Besides the stereo images and the depth map estimation, the ZED SDK also has some in-built algorithms to post-process the stereo data into meaningful results. The most relevant features of the ZED camera that are related to the VO are the following.

D.4.1 Positional Tracking

The ZED SDK provides the ego-pose using visual tracking for all the ZED cameras. In the case of ZED M or other models with inertial measurement unit (IMU) (refer table D.2), it fuses their data to provide more accurate pose estimates.

D.4.2 3D reconstruction

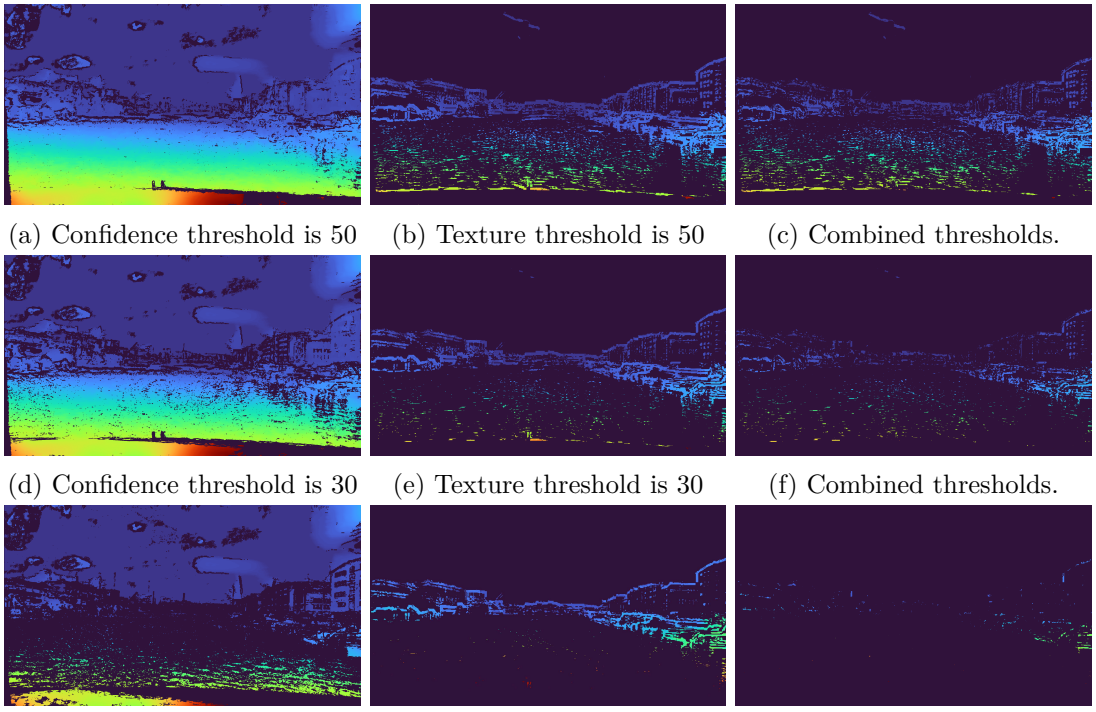
ZED SDK can also provide a 3D map of the environment. The map can be either a mesh or fused point cloud.

D.5 Multiple Camera Setup

It is more informative to have cameras looking sideways from the longitudinal direction of the boat. It will provide full coverage of the surroundings and multiple cameras can be used to jointly estimate the motion.

According to the ZED SDK, it is possible to set up multiple ZED cameras on one system or on different machines. For the multiple-camera setup, synchronization of the frames coming from different cameras is very important however ZED SDK doesn't talk about hardware synchronization of multiple ZED cameras. The key findings about the multiple ZED cameras are the following

- Two ZED cameras are not synced with each other, they most likely work all on their own even if they are connected with individual USB controllers to overcome bandwidth issues.
- The timestamp generated by ZED SDK is not the exact time when the camera captures the image, instead, it's when the image data is received by the PC and appears on the PC memory. So, there's no way we can recover the exact time point of image capture and therefore, the motion can not be compensated completely because of this asynchronization.



(a) Confidence threshold is 50 (b) Texture threshold is 50 (c) Combined thresholds.

(d) Confidence threshold is 30 (e) Texture threshold is 30 (f) Combined thresholds.

(g) Confidence threshold is 5 (h) Texture threshold is 5 (i) Combined thresholds.

Figure D.4: Comparison of different confidence and texture thresholds on the depth images computed with Ultra sensing with STANDARD mode with a depth range of 10m to 200m.

As mentioned above, hardware synchronization is not possible for multiple ZED camera models except for the ZED X model. If multiple ZED X cameras (up to 4) are integrated with the ZED box, hardware synchronization is possible.

Implementation of the Graphical User Interface (GUI)

Contents

The GUI is developed in Python language using the QT library [1]. It is slower than the former approaches as it is more complicated and requires the conversion of the images from OpenCV format to QT format, however, it is very intuitive to interact with and simple to integrate in any kind of application. In this section, I will talk about the different features offered by the GUI.

E.1 General Features of the GUI

The default features of the GUI for Sequencer 2.0 are highlighted in figure E.1. Each feature is labeled with a number in the figure E.1 and their explanation is given below.

1. **Dataset type Selection:** It is possible to select different kinds of datasets. There are three options that are available to the user. The user can switch between the ZED, KITTI, and VIDEO (not implemented yet) dataset formats.
2. **Dataset Files Selection:** Each of these dataset requires different input files. For ZED, the required dataset file should be in *SVO* (Dataset format used by the ZED SDK) format. For KITTI, the required parameters are the path to the left and right images and the path to the calibration file. Other optional parameters can also be used such as pose file,

[1] The link to the library is here

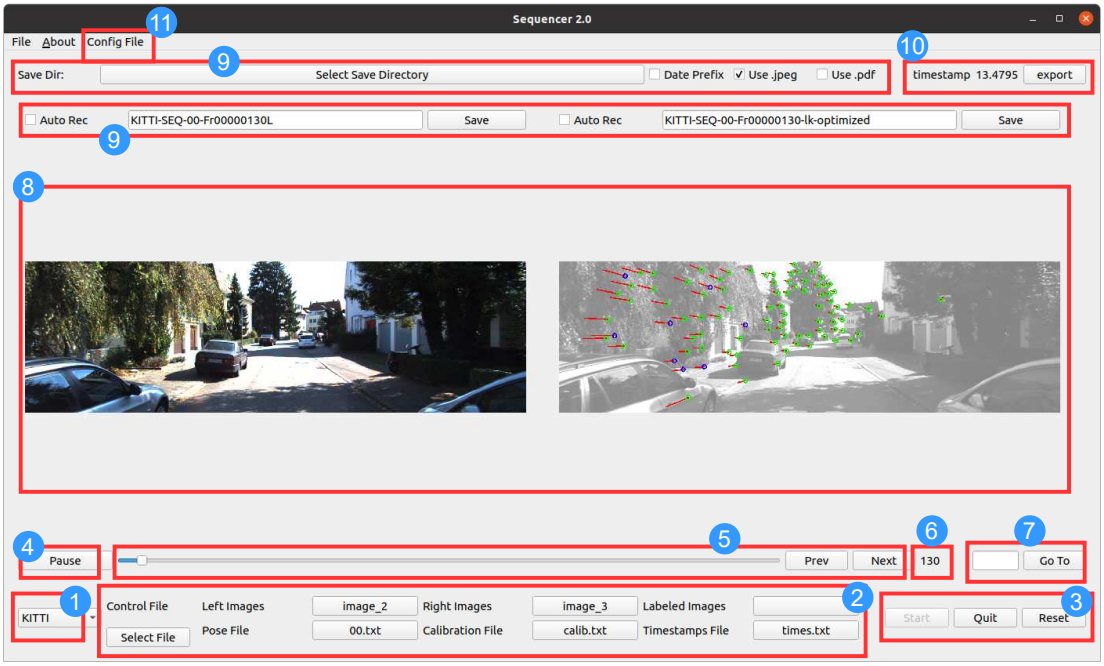


Figure E.1: Highlighted Features of the GUI

timestamps, etc. Any kind of dataset file can be passed to this dataset type as long as the format of the calibration file is similar to the one in KITTI. The third option is the VIDEO which can accept the video files to load the images from and the calibration file. The VIDEO mode (not implemented) is not interfaced with the sequencer yet. The option to upload the control file is available for every dataset type.

3. **Start/Quit/Reset:** The system can be started with the start button. When the *Start* button is pressed, the sequencer starts loading the image and starts rendering it. The *Quit* button is to quit the application. The *Reset* button (not implemented) is to reset the GUI.

4. **Play/Pause:** The beginning state of the player is *Paused*. In this state, the sequencer doesn't provide new images to the GUI and the GUI keeps rendering the first image. In the *Playing* state, the sequencer keeps providing the next image and the GUI renders it on the screen.

5. **Video Controller:** The GUI can be used as a video player for the sequencer. Similar to any video player, the user can jump between frames using the slider and use the *Next* and *Prev* buttons to move to the next and previous frame respectively.

6. **Frame Number:** The frame number of the rendered image is displayed here.

7. **Jump to:** For advanced usage, it is possible to enter the frame number directly and jump to this frame using the sequencer.

8. **Image Viewer:** The imagesare displayed here. There are two image windows by default.

9. **Save Results:** For the offline analysis of the system, sometimes it becomes necessary to save the processed images at particular instances. From the GUI, the user can select

the directory in which the results should be saved. He can also select if the current date should be added in from the name of the image to be saved followed by the format of the image. The Sequencer supports PDF and JPEG formats to save the image.

For each image window, there is a name assigned to the frame given by the system. This name is used to save the image when the *Save* button is clicked. Experimental feature: If the user wants to use some other name for the image, they can directly enter the name in the image name window and save it.

It is also possible to save all the images during the runtime of the GUI. *Auto Rec* option can be selected and the GUI starts saving all the images in the save directory.

10. **Timestamp:** The timestamp for the left and the right image is displayed here. These timestamps can be exported (Experimental) into a text file with the name *times.txt* in the save directory.
11. **Config Files Reader:** The state of the GUI such as the dataset file paths, the path to the save directory, etc. can be recorded into a *.yaml* file. This file when loaded again in the GUI will auto-fill the details and saves time.

E.2 Advance Features of the GUI

There are also some advanced features offered by the GUI and require the system to interface with the GUI in a more complicated way. These features are the following.

1. **Multiple Image Windows:** It is possible to visualize up to 3 images.
2. **Plotting:** A plot can also be drawn in real-time in the same GUI window.
3. **Dynamic hyperparameters:** System parameters can be changed from the GUI in real-time.

E.3 The interface of the GUI

The GUI is a standalone application, however, it needs an instance (Python Class object) of the sequencer. The sequencer class should have the following functions such that the GUI can call them and interact with the sequencer.

- *get_next_stereo_images*: It returns the stereo data for the next time step.
- *jump_to*: It moves the index to a particular frame number in the dataset such that when *get_next_stereo_images()* is called, the data belonging to this frame number should be returned.
- *get_frame_count*: It returns the total number of frames in the dataset.
- *close*: This closes the camera in case of a live ZED camera or performs other tasks.

The flow of execution of processes is shown in Fig. E.2. The steps are listed below.

1. When the user starts the system, it opens the GUI, passes an object of the sequencer to the GUI, and initializes the callback functions. The major callback functions are *on_start*, *on_streo_data*, *return_image_1*, and *return_image_2*. The system defines the behavior of these callbacks. These functions will be discussed in the next steps.

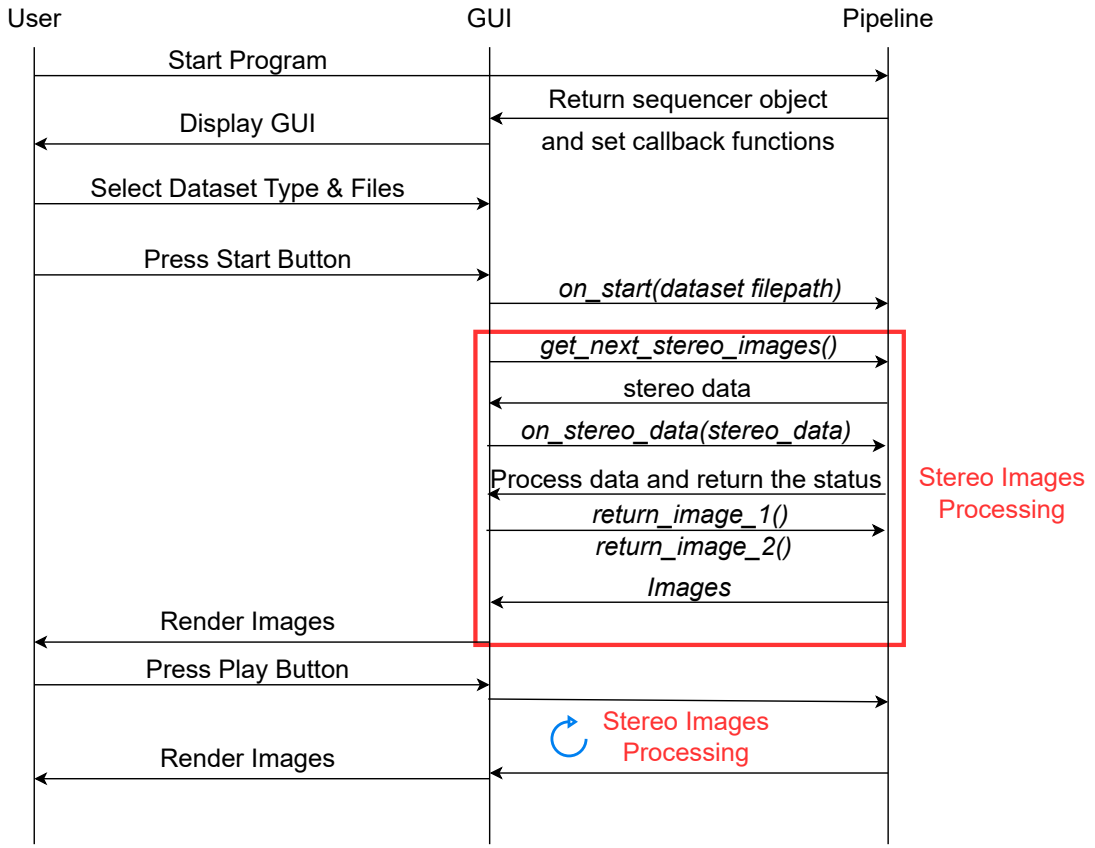


Figure E.2: Flow of execution of steps performed by the user, the GUI, and the Ferry-SLAM system.

2. Once, the GUI is rendered, the user selects the dataset type, for example, ZED, and selects the SVO file to be loaded.
3. After pressing the *Start* button, the GUI invokes the *on_start* function and passes the dataset information to the sequencer via the system. In the case of the ZED dataset, the sequencer only sets the runtime parameters for the SDK and allocates the memory to store the stereo data for a single instant only. For the KITTI dataset, it loads the calibration file, timestamps file, and the pose file (if available). It also processes the image files but doesn't load them in the memory.
4. Once, the sequencer object is initialized, the GUI requests stereo data from the sequencer by invoking the *get_next_stereo_images* function. The handler loads the images in the memory and returns the stereo data.
5. On successful retrieval of data, the GUI invokes the *on_stereo_data* callback. In this step, the stereo data is passed to the system for processing.
6. Once the processing is finished, the processing status is returned to the GUI. After that, the GUI requests the system to return the images to be displayed on the left and right image windows by calling the function *return_image_1* and *return_image_2* respectively.

During this, the system returns the images it wants to render and the name of the image to be displayed.

7. Once, the images are retrieved from the system, the GUI renders them.
8. The GUI goes to an infinite loop where it keeps following from 5 to step 7 until the *Play* button is pressed.
9. If the *Play* button is pressed, it repeats the step 4 to step 7 in a loop until *get_next_stereo_images* returns an *END_OF_FILE* signal or the system enters into a failure state or the *Pause* button is pressed.
10. The user press *Quit* button to exit the system and close the GUI.