Integrating Learned Motion Predictions

Into Factor Graph Optimization for Robot Pose Estimation

Candidate Number: KMWS7[1]

Computer Science, Robotics and Computation

Julier Simon
Lu Ziwen

Submission date: 14 February 2024

Abstract

SLAM, also known as Simultaneous Localization and Mapping, is essential for autonomous robots in practical applications as it allows them to comprehend and explore their surroundings in real-time. This technique is essential for enabling robots to carry out activities in unfamiliar environments by enabling them to generate a map of their surroundings while concurrently monitoring their own position within that map. Currently, deep learning techniques have gained widespread use in this domain because to their ability to effectively analyse intricate scene data and achieve accurate long-term predictions, even in situations where the underlying process and observation noise are uncertain and difficult to define. This work presents a technique that improves the accuracy of robot pose estimation by incorporating deep learning-based motion and observation predictions into factor graph optimisation for SLAM systems. The significance of motion prediction in autonomous driving vehicles is underscored, as it is vital for enhancing the robot's ability to design and locate safer and more efficient pathways and speeds. Additionally, this study will introduce using Levenberg-Marquardt (LM) optimiser, which is an algorithm used to solve non-linear least squares problems in the field of robotics for the efficient solution of the SLAM problem.

Contents

1 Introduction **2**

1.1 Background and Motivation 2

1.1.1 Overview 2

1.1.2 Problem statement 3

1.2 Research Framework and Objectives 3

1.2.1 Aim and approach 3

1.2.2 Thesis outline 4

1.3 Mathematical Notation 4

2 Literature Review **6**

2.1 Stages in SLAM Development 6

2.2 State Estimation Technique 7

2.2.1 Filtering 7

2.2.2 Smoothing 7

2.3 Combining Learning and Smoothers 8

2.4 Optimization Algorithm 9

2.5 Related Work 10

3 Background & Preliminary Work **11**

3.1 Visual SLAM Framework 11

3.1.1 Loop closing 12

3.1.2 Optimization 13

3.2 Maximum A Posteriori 13

4 Mathematical Formulation of the SLAM Problem **15**

4.1 Mathematical notation 15

4.1.1 Covariance 15

4.1.2 SLAM notation and equation 16

4.2 Bayesian networks 17

4.3 Maximum a Posteriori Inference 19

4.4 Factor graph 20

5 Methodology **22**

5.1 State Estimation in Factor Graph Structure 22

5.2 Learning Robot Motion 24

5.2.1 Introduction to robot motion modeling 24

5.2.2 Loss function 25

6 Implementation & Experiments **27**
6.1 Library Requirement . . . 27
6.2 Assumptions and Noise Model Setup . . . 27
6.3 Trajectory simulation . . . 28
6.3.1 Landmark Generation . . . 30
6.3.2 Dynamic model . . . 31
6.3.3 Observation model . . . 33
6.4 Data visualization . . . 34
6.5 Data saving . . . 34
6.6 Deep Neural Network . . . 34
6.6.1 Data collection . . . 34
6.6.2 Data preprocessing . . . 35
6.6.3 Neural Network Architecture . . . 35
6.7 Training . . . 36
6.7.1 Batch size . . . 37
6.7.2 Learning Rate . . . 37
6.8 Testing . . . 38
6.9 Building Factor Graph by GTSAM Python wrapper . . . 38
6.9.1 Variables & Key . . . 38
6.9.2 Noise model . . . 39
6.9.3 Factors . . . 39
6.10 SLAM System . . . 40
6.10.1 Optimization techniques in SLAM . . . 40
6.10.2 Integrating deep Learning for enhanced pose estimation . . . 41
6.10.3 SLAM simulation . . . 41

7 Results & Discussion **43**
7.1 Advancements and Challenges in SLAM Systems . . . 43
7.1.1 Impact of wheel speed range on training performance . . . 43
7.1.2 Evaluating the Impact of Neural Network Integration on SLAM Accuracy . 44
7.2 Limitations . . . 47

8 Conclusion **48**

A **52**
A.1 Code Listings . . . 52
A.2 Additional Table . . . 64
A.3 Gauss-Newton . . . 65

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Overview

The domain of robotics and autonomous systems has experienced significant progress. Autonomous driving vehicles have a significant impact on road safety, traffic congestion, energy consumption, and other factors that contribute to the sustainability of transport systems [18]. An autonomous vehicle can be categorised into four components: environment perception, motion prediction, motion planning, and motion control [19]. Among them, the motion prediction element is crucial for ensuring the long-term viability of autonomous driving cars. Nevertheless, the implementation of autonomous driving on a wide scale is still far. One significant factor is the considerable uncertainty that vehicles or robots encounter in complex situations, such as the unpredictable actions of an item in the near future. This uncertainty, such as process or observation noise affecting a system's state evolution, makes it difficult for vehicles to anticipate well and can result in uncomfortable or imprecise brake control. Such disturbances can stem from external factors, hardware limitations, or model inaccuracies. Motion prediction modules in driving cars aid in enhancing the vehicles' comprehension of the dynamic world around them. This is achieved by forecasting the future motion states of the vehicle itself based on control inputs like wheel speed. The active braking system in autonomous driving vehicles can properly evaluate the chance of collision by considering future states and nearby landmarks, thereby preventing vehicle collisions.

Furthermore, there have been notable breakthroughs in simultaneous localization and mapping (SLAM) within the auto industry, which have greatly contributed to the progress of autonomous driving technology. SLAM allows for the simultaneous creation or update of maps while tracking a robot's position inside those maps [8]. Among several features in SLAM system, state estimation is crucial for maintaining a precise and consistent image of the system's condition. It involves combining information regarding position, speed, orientation, and other important dynamics that are necessary for navigation or accurately determining the location of an object.

This study aims to investigate two areas: robot motion and state estimation in the SLAM system. The robot's motion plays a crucial role in the driving system by receiving information from sensors and control inputs. It then uses this data to predict the surrounding environment for motion planning and localization in the SLAM system. This ultimately helps the vehicle or robot plan safer and more efficient paths and speeds.

1.1.2 Problem statement

The aforementioned challenges briefly present the fact that the widespread adoption of autonomous vehicles is currently limited. The rationale is that traditional methods for addressing state estimation with the defined noise model involve employing Kalman filters, Particle filters, and optimization-based smoothers, as explained in Section.2.2. In many applications, developing models for these estimations is analytically uncomplicated (Sec.6.3.2 & Sec.6.3.3), though these physics-based methods are computationally fast but cannot consider complex scene factors due to the difficulty of modelling the process and observation model in the real system. For example, within a simple environment where pose estimation uses inertial measurement units, it is feasible to build a simple correlation between the data obtained from gyroscopes, accelerometers, and magnetometers in order to infer the approximate position and orientation of the robot [38]. The uncertainties of these systems are commonly expressed as Gaussian distributions with manually adjusted, diagonal covariance matrices, which enables an accurate approximation.

Nevertheless, whereas standard probabilistic techniques such as filters and smoothers are effective in managing less intricate systems and shedding light on uncertain components, they frequently prove inadequate for more intricate robotic tasks. Systems that depend on data-rich modalities such as pictures, sound, or tactile feedback are fundamentally challenging to model because of their complex dynamics and diverse noise circumstances. External factors, such as variations in illumination or obstacles, can additionally limit the precision of condition assessments. For instance, the accuracy of an object's position estimated from an image can vary dramatically, from being highly accurate under ideal conditions to being entirely unreliable when obstructed or in poor lighting. Given these challenges, there is a growing need for more resilient estimating methods. Deep learning, with its capacity to adaptively learn from both dynamic and observational data, offers a promising solution. It holds the potential to provide precise and efficient robot motion in state estimation across a range of complex robotic environments, adapting to their dynamic nature and ensuring reliability even under challenging conditions.

1.2 Research Framework and Objectives

1.2.1 Aim and approach

This study focuses on exploring the integration of deep learning techniques with factor graphs for robot state estimation. Drawing inspiration from the research of Yi et al. [38], our approach emphasizes learning the relative transformation from the control input. By learning the information regarding the estimation of relative displacement and the noise covariance from a neural network, these predictions would then be fed into a factor graph for final graphical optimization in order to estimate the global robot trajectory. Furthermore, we incorporate the model to acquire understanding of the robot's poses in order to predict the observation between the landmark and the robot. This ensures that the entire system aligns with the SLAM system. The primary aim is to develop a robot motion model that leverages deep learning and enhances localization accuracy through the application of factor graphs.

In our research, we have adopted factor graphs, a form of bipartite graph [6], as the primary methodology for state estimation, motivated by their ability to effectively merge the benefits of deep learning and probabilistic graphical models. In factor graphs, vertices represent variables or events, and edges indicate probabilistic constraints based on measurements or prior information. These elements collaborate to infer subsequent states via observation and process models.

In addition to adaptability combining deep learning, the preference for factor graphs, especially in contrast to conventional filtering methods for state estimation (referenced as 2.2.1), is justified by several key advantages. These include their global optimization, proficiency in managing non-linear dynamics, scalability and ability to handle sparse data structures, and effectiveness in addressing loop closures. Each of these aspects aligns with the objectives of the SLAM study and also greatly decreases computation time. For a more comprehensive explanation of these reasons and their implications, we direct the reader to the Literature Section (2.2.2), where an in-depth analysis and discussion are presented.

In conclusion, the analysis and discussion presented in this research aim to evaluate the efficiency and accuracy of integrating deep learning models with factor graphs, as opposed to solely utilizing factor graphs for state estimation. This approach enables an understanding of the simulator's behaviour by training a model explicitly to predict robot motion, observation, and covariance. This estimation will then be utilized in our factor graph assessments. In essence, we're synergizing deep learning methodologies with optimization strategies. This combination promises accuracy and enhanced resilience in the results we aim to achieve.

1.2.2 Thesis outline

The thesis starts by highlighting the importance of state estimation within SLAM. This introduction paves the way for a detailed literature review that focuses on modern state estimation algorithms. This review not only discusses the development of smoothers and filters (Sec.2.2) but also dives into other key optimization techniques, as well as integrating deep learning technique. It then smoothly transitions to explore other related studies in the field. In the section on preliminary work, we delve into VSLAM (Sec.3.1) to better grasp the core concepts of SLAM. This exploration makes it clear that our main focus is on the back-end details of SLAM, especially state estimation, loop closure, and optimization. Next, we dive into the mathematical side of things, emphasizing the role of the factor graph and the maximum a posteriori (MAP) estimate (Sec.4.3) in understanding the factor graph. The methodology chapter 5 outlines the research process, starting with the introduction of factor graph for state estimation and the assumptions made in this study to understand the robot's movement and observation. Subsequently, commence the experiment setup gathering data from trajectory simulation and acquire information regarding the robot's movements through the labelling in the training phase. Afterwards, this model is smoothly incorporated back into the factor graph. The discussion and results chapter offer a practical perspective, discussing the actual implementation, experiments conducted, and a review of the outcomes. The thesis wraps up with a conclusion that summarizes the major discoveries and the limitations of the research.

1.3 Mathematical Notation

Throughout this dissertation, the following conventions and notations are consistently used: Scalars are presented in italic font (e.g., a), whereas vectors and matrices employ bold font (e.g., $\mathbf{a}$ and $\mathbf{A}$). Hollow bold symbols represent special sets, such as the set of real numbers, $\mathbb{R}$.

Starting from **Chapter 4**, an extensive use of mathematical symbols is introduced. A brief overview is presented here for clarity:

- $\mathbf{x_k}$: Represents a state with x, y coordinates along with a bearing angle at time k.
- $\mathbf{m}$: Denotes a landmark position consisting of coordinates x^i and y^i.

- $\mathbf{X}$: A set containing all states to which $\mathbf{x}$ belongs.
- $\mathbf{M}$: A set comprising all the landmark positions.
- $\mathbf{Z}$: A set that includes every $\mathbf{z}$ measurement.
- f(.): Process model
- h(.): Measurement model

Readers are encouraged to refer to this section if any confusion arises regarding the employed notations in subsequent chapters.

Chapter 2

Literature Review

This thesis presents an experimental investigation into the back-end system of Simultaneous Localization and Mapping (SLAM), focusing specifically on state estimation and optimisation. The literature review will clarify the evolution of state estimation solutions in SLAM and examine the methodologies that support this aspect. While state estimation approaches have increasingly shown success in static environments, their effectiveness in more complex and dynamic settings remains limited. This limitation has led to an emerging trend of applying deep learning methods, known for their advancements in computer vision, to back-end SLAM systems. This integration highlights the potential of machine learning, particularly deep learning, to enhance the accuracy and efficiency of state estimation, though it introduces complexities due to process and observation noise. Special attention is given to the Differentiable Bayesian Filter and a novel approach that combines trainable parameters with factor graphs. The aim of this literature review is to provide a structured synthesis of the key breakthroughs and discoveries that form the foundation for the methodologies used in our study.

2.1 Stages in SLAM Development

The evolution of the SLAM technique can be simply described in two primary stages at this point. The initial stage, from the late 1980s to the mid-2000s, focused on resolving the SLAM problem through the application of Bayesian filtering techniques. Pioneering research by Smith et al.[32] systematically represented the SLAM problem as a stochastic estimation problem. In this approach, the robot's pose and landmarks were treated as random variables following certain distributions. By utilising motion and observation data, filtering theory was applied to predict system states and update observations. This method enabled map updates with real-time performance. Notable algorithms from this phase include the Kalman Filter, the Extended Kalman Filter, and the Particle Filter [11, 34]. These filtering approaches make the assumption of Markov characteristics in state estimation, relying solely on information from adjacent frames to estimate the robot's state. This makes it challenging to address the data association problem between current and historical frames. In SLAM systems, the motion and observation equations are nonlinear functions. The filtering method figures out the posterior probability of the state using a first-order Taylor approximation. This means that when there are strong nonlinearities, it is impossible to avoid making large linearization estimation errors. Additionally, uncertainties in system parameters and observations can accumulate errors, resulting in inconsistencies in map building. In terms of implementation, filtering methods require storage, maintenance, and updates of the mean and variance of state variables, and the storage capacity increases quadratically with the number of estimated state variables. Consequently, filtering methods are only suitable in scenarios with limited computational resources or fewer estimation variables.

During the period from the mid-2000s to the mid-2010s, the second phase of research on SLAM primarily concentrated on the theoretical examination and practical execution of SLAM, with extensive research on the observability, convergence, and consistency of SLAM problems. A key aspect of SLAM applications during this period was reducing the demand for computational resources while achieving real-time performance and expanding the scale of application environments [9]. The use of graph optimization techniques [7] to construct nonlinear least squares objective functions for observation equations became mainstream in SLAM research. To do this, the robot's pose and landmarks were used as variables that needed to be optimized. Iterative estimation methods, such as the Newton method and the Levenberg-Marquardt algorithm, were used to improve the accuracy of local positioning and landmark estimation. Additionally, the recognition of the sparsity in SLAM problems gradually emerged, making it possible to obtain globally consistent solutions.

2.2 State Estimation Technique

2.2.1 Filtering

The filtering concepts have served as the basis for state estimation over the years [26]. Filtering approaches primarily focus on developing and upholding a probabilistic perception of the current state of a system. As the simulation progresses, this belief is iteratively refined based on the complexities of dynamic (process) and observation models. This fundamental problem, for example, is typically addressed using methods such as the Kalman Filter (KF) [36].

Nevertheless, real-world robotic systems are frequently complex and nonlinear, comprising sensors, dynamics, and noise that are difficult to predefined or adjust. Although the Kalman filter produces excellent results in linear systems, there are issues with computational and storage costs [2]. As a result, a modified Kalman filter, Extended Kalman Filter (EKF) and Particle Filter (PF) solve the computation cost and often deal with the non-linearity of most real-world systems [11, 34]. In systems that do not strictly conform to linearity, using approximations becomes essential to assist the filtering process. Moreover, one of the most notable characteristics of filtering is its recursive nature. This guarantees that, despite a period, the memory requirements remain consistently stable for problems that exhibit a fixed state dimension. Within this organised framework, the perception recorded at a particular time step t is necessarily connected with, and indeed generated from, the perception at $t-1$. Moreover, the model considers relevant observations or control inputs at the given time step t. However, these filtering characteristics solely take into account the present and preceding states $(\mathbf{x_t}, \mathbf{x_{t-1}})$ and are unable to detect extended loop closures $(\mathbf{x_t}, \mathbf{x_{t-n}})$, even if the positions are same. Hence, we will introduce the smoothing technique for state estimation.

2.2.2 Smoothing

One distinguishing aspect of smoothing, as opposed to filtering, is its ability to retain past measurements comprehensively, allowing for the continuous recovery of whole state trajectories [7]. Precisely, it can estimate an unknown probability density function recursively over time using incremental incoming measurements. The process of filtering, which is commonly linked to recursive methodologies, occasionally shows constraints in terms of precision and computational time. In contrast, smoothing aims to infer the system's state at a prior point in time based on an entire set of observations throughout the time sequence, including those occurring subsequent to the specific moment of interest. This is why smoothing frequently yields a more precise estimation of past states compared to filtering, as it incorporates future data [31].

Recent research has proposed using graphical models to address the scalable and nonlinear problem for state estimation due to the large maps. The study, as indicated by Dallaert [5], has shown that structure-from-motion (SfM) has effectively addressed a problem similar to visual SLAM, which is a technique where a device uses visual input from a camera to simultaneously map an unknown environment and determine its location within that map. SfM is a technique used to reconstruct a three-dimensional (3D) model of the environment from a series of photos, typically taken by different individuals at different times with varying cameras. However, the estimation techniques employed in SfM are not filtering algorithms. Instead, they represent exceptional cases of an approach known as a "Factor Graph". In the meantime, these probabilistic graphical models have gained popularity in the field of SLAM within the domain of state estimation. Therefore, in this work, we focus on factor graphs providing a framework for visualising and solving smoothing problems and examining the complexities and evolving applications of smoothing.

2.3 Combining Learning and Smoothers

The exploration of incorporating algorithmic frameworks within deep learning techniques has been conducted for various robotic challenges. These include issues in state estimation [15, 20, 23], as well as in the areas of planning [35] and control [17]. A significant contribution in this field is [25], where a combination of several distinct differentiable algorithms forms a fully trainable network, termed "Differentiable Algorithm Network." This network is specifically designed to tackle the comprehensive task of guiding a robot to a target location within an unfamiliar environment. The potential of a differentiable filter is fully realized when combined with other advanced techniques. As an example, incorporating differentiable algorithms into a factor graph instead traditional filtering such as the particle filter. As explored in the work of [38], presents a method that can be used to learn smoothers that outperform both Long Short-Term Memory (LSTM) and differentiable filters. LSTM is a recurrent neural network (RNN) architecture well-suited for sequence prediction problems and time-series data. From these results, it becomes apparent why we have chosen the factor graph as our primary method. This technique allows for the simultaneous learning of both the state and noise models within an integrated structure.

Moreover, Czarnowski et al.[4] pioneered an approach wherein depth maps generated by deep neural networks play a pivotal role in computing factors like photometric, reprojection, and sparse geometric for a Monocular SLAM system. On a parallel front, Rabiee and Biswas [28] proposed the IV-SLAM methodology to guide feature extraction. IV-SLAM adopts a more holistic, context-aware approach by combining learning techniques with smoothers. This enables the system to identify and tackle errors emerging from complex visual scenarios effectively. The research data strongly supports the theory that IV-SLAM outperforms classic V-SLAM in various aspects. This is evident through its improved capacity to foresee errors, a significant reduction in tracking errors, and considerable resilience in the face of tracking failures in complex robotic environments. Moreover, Sodhi et al. [29] emphasised the importance of incorporating tactile measurements into a factor graph framework. The main objective of this approach is to estimate the poses of manipulated objects accurately.

2.4 Optimization Algorithm

The extraction of relevant information from these graphs is fundamentally dependent on the utilisation of efficient optimisation techniques. Over the years, numerous optimisation algorithms have been modified to operate effectively with factor graphs, thereby providing resolutions to particular difficulties within this framework. Among these models, the most common problem to be solved in optimization algorithms is the nonlinear least square equations that play a pivotal role in the simulation of physical phenomena. When the aforementioned models, which are initially formulated as nonlinear partial differential equations, undergo the process of discretization, they transform into extensive systems of nonlinear least square equations. Frequently, when closed-form formulations for nonlinear least square equation solutions are not readily available, iterative methods are employed to obtain numerical approximations. This approach generates sequences that converge towards the desired solution.

Newton's Method [27] is a classical optimisation technique that uses second-order information, notably the Hessian matrix, to locate the extrema of a function accurately. The traditional representation of Newton's paradigm possesses a greater level of generality, rendering it suitable for functions that exhibit qualities of smoothness. The Gauss-Newton method [30], derived from Newton's approach, is well-suited for addressing problems involving nonlinear least squares, which are commonly used within the framework of factor graphs. In each iteration, the approach applies a linear transformation to the residual functions inside the framework of the current approximation, and it subsequently solves the resulting linear system to refine this approximation. The proficiency of the algorithm in dealing with least squares has strengthened its position in the field of SLAM [37], and other problems centred around factor graphs. It offers the potential for rapid convergence, particularly when the first approximation closely resembles the true answer. Moreover, the Levenberg-Marquardt (LM) algorithm is widely recognised as a highly effective numerical optimisation method, particularly in the context of nonlinear least squares problems [10]. The robustness of the LM algorithm is frequently regarded as an enhancement compared to the Gauss-Newton method. This attribute arises from its ability to mitigate unstable behaviours in specific situations. By incorporating a damping component, this approach effectively combines the characteristics of the Gauss-Newton method with the gradient descent method [14]. The merging guarantees the stability of convergence, particularly in cases where the initial approximations deviate from the true solution or when the problem itself is inherently challenging.

The integration of factor graphs has been recognised as a robust technique for reliable state estimation in difficult terrains, leading to its widespread acceptance. In recent times, there has been a heightened emphasis on enhancing the differentiability of conventional approaches. Tang, Tan, and their colleagues [33], in conjunction with Jatavallabhula and others [21], have introduced differentiable modifications to the Levenberg-Marquardt optimisation technique. The increasing inclination towards differentiability serves as the basis for seamlessly integrating optimisation procedures into deep learning ecosystems, where the availability of gradient data is of utmost importance. In conclusion, the implementation of optimisation techniques in factor graphs has had a significant impact on disciplines such as robotics and computer vision due to their specific advantages and disadvantages.

2.5 Related Work

Recursive Bayesian filters, a collection of algorithms highlighted by [3], provide a method of probabilistic state estimation by integrating perception and prediction in a systematic fashion. The utilisation of these filters requires the implementation of an observation model and a process model, containing inherent noise components that evaluate the reliability of perception and prediction. Yet, dynamic models suitable for these estimators might be challenging to derive, mainly when dealing with high-dimensional inputs such as camera images.

In the past few years, this is why deep learning has emerged as a prominent approach for the analysis of sophisticated data. Recent research [23, 24] has also shown that integrating data-driven models, such as neural networks, into Bayesian filters for end-to-end training is feasible. The integration of a learning-based approach with conventional algorithms, such as Kalman and Particle filters, which are all Bayesian filters, has had a significant impact on the development of state estimation techniques [1]. The fundamental purpose of a differentiable filter is to possess the capability of being trained end-to-end, utilising the principles of backpropagation and gradient descent, which are widely employed in the field of deep learning. This attribute contrasts conventional filters such as the Kalman and Particle filters, which lack basic differentiability and cannot be easily optimised using gradient-based approaches. Recent literature, such as [15], has showcased backpropagation through the prediction and correction steps of the Kalman filter. In addition, Jonschkowski and Crock [20] evaluate the performance of a differentiable particle filter. The findings of the study [1] indicate that differentiable filters (DFs) based on the Kalman filter and Particle filter also frequently outperform unstructured networks such as LSTM networks in terms of performance.

Chapter 3

Background & Preliminary Work

This chapter explores the fundamental elements of simultaneous localization and mapping (SLAM) that are crucial to this study, employing visual SLAM as a demonstrative example. SLAM is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. The fundamental components of the visual SLAM framework include five essential features, namely: sensor information retrieval, visual odometry, loop closing, optimisation, and mapping. A brief review of these components is presented. Following this, the chapter provides a comprehensive examination of the back-end components of SLAM, with a particular focus on loop closure, optimisation, and the technique addressing SLAM problems, as they play a crucial role in the smoother of this research.

3.1 Visual SLAM Framework

Prior to entering into the complex field of SLAM, it is essential to establish a strong foundation. To achieve this objective, our investigation will start with a study of visual Simultaneous Localization and Mapping (VSLAM). This approach will offer a comprehensive understanding of the fundamental framework and constituents of SLAM and reveal its importance and wide-ranging applications. By understanding visual SLAM, we'll gain a perspective on what SLAM truly is and the transformative impact it has on various fields. The entire visual SLAM process (Figure.3.1) can be divided into the following steps [13]:

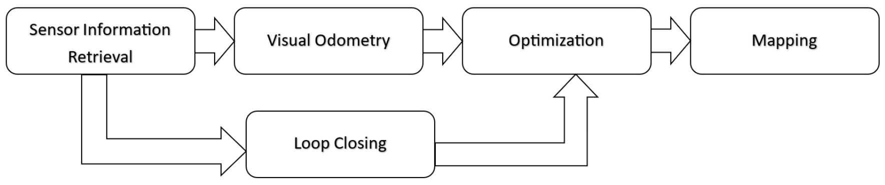

Figure 3.1: Flowchart of Visual SLAM

1. Sensor Information Retrieval: Visual SLAM primarily involves reading and preprocessing camera image data. For robots, this might also include reading and synchronizing data from encoders and inertial sensors.

2. Visual Odometry (VO): Visual odometry is responsible for determining the motion of the camera between consecutive images and constructing a local map based on visual cues. This

process is often referred to as the “Front-End”.

3. Loop Closure Detection: This process identifies if the robot has revisited a specific location. Upon detecting a loop, the relevant data is forwarded to the back-end for further refinement.

4. Back-end Optimization: Following the front-end’s estimation, the back-end receives camera poses from different timestamps and data from loop closure detection. It then refines these inputs to produce a globally consistent trajectory and map, hence its designation as the “Back-End”.

5. Mapping: Utilizing the refined trajectory, a map is generated that aligns with the specific objectives of the task.

The framework and its related algorithms have undergone extensive standardisation and are widely accessible in numerous visual and robotic software libraries. By employing these techniques, it becomes possible to construct a SLAM system that exhibits the ability to do real-time localization and mapping inside typical working situations. Therefore, when the working environment is restricted to static, inflexible settings with minimum variations in lighting and no human intervention, the SLAM system can be deemed highly developed.

In the following discussion, we will explore the distinct tasks associated with the back-end module for the purpose of this study, known as loop closing and optimisation. However, comprehending the operational principles of these entities requires a basic grasp of mathematical concepts, which will be explored in Chapter.4.

3.1.1 Loop closing

In the realm of robotic navigation, consider a scenario where a robot, after traversing a certain path, returns to its initial position. Nevertheless, due to drift in its sensors or algorithms, the robot’s estimated position might not align with its actual starting point. This discrepancy is primarily because not all sensors are perfect; some may have inherent drift due to unmodeled appropriateness in their analytic models, which manifests as time-varying biases in the data they produce. Within the framework of SLAM, it is known that if a robotic system possesses the capability to discern the event of “revisiting its initial position” or to identify the “point of origin accurately” and subsequently recalibrate its positional estimate to this reference point, it can effectively mitigate the accumulated drift [22]. This fundamental process of identification and recalibration is represented as the core principle of loop closure detection.

Loop closure detection is closely related to both “localization” and “mapping”. In fact, the primary utility of a map in SLAM is to provide the robot with a memory of its traversed locations. To facilitate loop closure detection, the robot must possess the capability to recognize previously encountered environments. Various methodologies can be employed for this purpose. For instance, one could place a distinct marker, such as a QR code, in the robot’s environment. Upon detecting this marker, the robot would recognize its return to a known location. However, this approach essentially introduces an external sensor into the environment, which might not always be feasible or desirable. A more autonomous approach would use the robot’s onboard sensors, particularly its visual sensors, to achieve this recognition [16]. By assessing the similarity between captured images, the robot can infer revisits to prior locations. Successful loop closure detection can significantly reduce cumulative errors. Therefore, visual loop closure detection is essentially an algorithm that calculates the similarity of image data. The richness of image information makes it easier to detect loops correctly. After detecting a loop, we inform the back-end optimization algorithm with information received from the VO, such as “pixel 1 and pixel A are identical”. Subsequently, the back-end then refines the robot’s path and map to align with the loop closure data. Consequently, with accurate loop closure detection, it becomes feasible to rectify accumulated errors, resulting in a globally consistent trajectory and map.

3.1.2 Optimization

The optimisation method holds great importance in enhancing estimated trajectories and produced maps. As a robot or imaging device navigates, it continuously accumulates unprocessed data in the form of measurements and observations. However, it should be noted that these measurements are susceptible to noise and inaccuracy, even in high-quality sensors, which are also vulnerable to external influences such as magnetic fields or changes in temperature, unavoidably demonstrate noise. Hence, in addition to the task of calculating camera motion based on images, it is essential to take into account the amount of noise in these data received from VO, its temporal propagation, and the level of confidence associated with the present estimate. This will introduce the issues related to optimising the back-end, which involves the estimation of the system’s overall state based on noisy data and determining the uncertainty associated with this prediction. This process is commonly referred to as Maximum-a-Posteriori (MAP) estimation. It will be briefly explained in Section 3.2, and more detailed information, including mathematical equations, will be provided in Chapter 4.

3.2 Maximum A Posteriori

The literature review (Sec.2.2) outlines the benefits of using a factor graph structure in SLAM systems and points out the drawbacks of traditional filtering methods. This section aims to tackle the issue of including historical data in SLAM by introducing density estimation as a key solution provided by the factor graph for calculating the probability distribution of a series of observations in a specific problem area. Typically, estimating the full distribution is not practical; therefore, a practical approach is preferred, focusing on obtaining the expected value of the distribution, such as the Gaussian distribution. The Maximum a Posteriori (MAP) algorithm, based on Bayesian inference, is presented as an effective method for estimating the distribution and model parameters that most accurately reflect the observed dataset. For example, considering a set of observations ($\mathbf{Z}$) from a domain ($\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_n$), it is assumed that each observation is independently and identically distributed following the same probability distribution. Density estimation thus involves choosing a suitable probability distribution function and identifying its parameters to best represent the joint probability distribution of the observed data ($\mathbf{Z}$). While there are various methods for solving this problem, this discussion primarily focuses on two common approaches: the Bayesian method of MAP and the frequentist method of Maximum Likelihood Estimation (MLE). This study will not explore the details of MLE; however, it’s important to note that the key difference between MLE and MAP is the latter’s use of prior probability over the distribution and parameters. This addition of prior knowledge is the reason for choosing the MAP technique for our study, especially due to the assumption of the initial state of the robot in the map.

Utilizing the Maximum A Posteriori (MAP) estimate offers a solution to the SLAM problem. It determines the most probable configuration of states, or robot poses, based on the control input and prior information. The particulars of the mathematical equation will be explored in Chapters

4 and 5. As indicated in eq.(5.11), this approach is underpinned by a quadratic cost function. This function's utility emerges from several features: Firstly, the summation that spans from 1 to k guarantees the incorporation of all past measurements, ranging from the beginning to the current timestep. Such an approach means that the optimization goes beyond the immediate state, examining the entire historical data, thus ensuring consistency. Secondly, the term $\mathbf{Q}_k^{-1}$ represents the inverse of the noise covariance matrix (information matrix) at the $k-th$ time point. By weighting the error by this term, the optimization accounts for the uncertainty in measurements. This means measurements with high uncertainty will have less influence on the optimization, while more certain measurements will have a stronger influence. Moreover, when the SLAM problem is represented using a factor graph, the optimization problem naturally breaks down into smaller sub-problems. Each factor in the graph corresponds to a constraint or measurement, and the structure of the graph provides a visual and computational way to understand and solve the nonlinear problem. In essence, by optimizing this cost function using the MAP estimate, the SLAM problem can efficiently deal with past measurements, incorporate noise models, and find the most probable path and map given the data. The factor graph representation aids in breaking down this complex problem into more manageable parts, making the optimization feasible and more intuitive.

The Maximum A Posteriori (MAP) estimates are derived through an optimization procedure. For fully linear systems, it can be demonstrated that the MAP estimate is directly solvable using matrix inverses. However, most systems are inherently nonlinear. As such, the task becomes one of nonlinear least squares estimation. Due to the uncommon occurrence of closed-form solutions for nonlinear systems, an iterative optimisation method such as Gauss-Newton (appendix.A.3) is employed in this research to ensure convergence to the optimal solution.

Chapter 4

Mathematical Formulation of the SLAM Problem

From the previous chapter, a foundational understanding of the structure and core functionalities of the SLAM system's modules should have been established. This research narrows its focus to the back-end aspect of SLAM, with particular emphasis on state estimation and optimisation. However, for the development of functional programs, mere intuitive understanding is insufficient; therefore, it becomes imperative to represent the SLAM process rigorously using mathematical formulations.

In this discussion, three primary components will be addressed: mathematical concepts for SLAM, Bayesian networks, and factor graphs. Initially, the mathematical notation and motions employed to represent the system are elucidated. Subsequently, the emphasis shifts to Bayesian networks, the foundational probabilistic models essential for SLAM. In real-world scenarios where sensor data is often readily available, the transformation of Bayesian networks into factor graphs is vital; hence, this conversion process by Maximum a Posteriori Inference (MAP), based on the sensor data, will be detailed.

4.1 Mathematical notation

4.1.1 Covariance

Estimators often return a point estimate of a quantity of interest; however, we often need to know how accurate this estimate is. The way of quantification of uncertainty is to accompany every estimated state with a covariance matrix.

However, it is important to consider the consequences of this matrix and understand its significance in the estimating process. At its core, variance measures the spread or distribution of a set of data points. In the context of estimation [22], if we're estimating a single value (like the position of a robot along a straight line), its variance will give us an idea of the uncertainty or confidence in that estimation. Formally, the covariance is given by [22],

$$\operatorname{cov}(x_t) = \mathbb{E}\left[(x_t - \mathbb{E}[x_t])^2\right] \tag{4.1}$$

and the standard deviation is the square root of this

$$\mathrm{std}\,(x_t) = \sqrt{\mathrm{cov}\,(x_t)} = \sqrt{\mathbb{E}\left[(x_t - \mathbb{E}\left[x_t\right])^2\right]} \tag{4.2}$$

In numerous practical applications, especially in fields such as robotics or spatial analysis, it is imperative to estimate multiple interrelated quantities concurrently, such as position coordinates in the x, y, and z directions. In these cases, knowing just the variances (or uncertainties) in individual estimates is not enough; it is equally significant to know how these estimates relate or vary with each other. This necessitates the introduction of covariance, which quantifies the joint variability between two variables. Therefore, for example, the cross-correlation is computed from [22],

$$P_{xy} = \mathbb{E}\left[(x - \mathbb{E}[x])(y - \mathbb{E}[y])\right] \tag{4.3}$$

When dealing with multiple variables, the covariance (and variances) of all possible pairs of variables are combined into a matrix format known as the covariance matrix. The diagonal entries of this matrix represent the variances of the variables, and the off-diagonal entries represent the covariance between the variables. The covariance matrix is a thorough representation of data distribution and correlations. Its further elaboration will be provided in the subsequent discussion on the Gauss distribution.

4.1.2 SLAM notation and equation

In an environment where a robot navigates unfamiliar 2D terrain with a specific sensor, the camera typically captures data at distinct time intervals. To frame this situation mathematically, it's crucial to note that the robot's continuous motion can be thought of as a series of discrete steps, represented by timestamps $t = 1, 2, ...k$. The robot's position over time $\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_k$ collectively form its trajectory, and this progression is often referred to as the robot's "platform state", evaluated on a step-by-step basis. In addition, each state might comprise several pieces of information, such as the robot position and the bearing angle in the map. As an illustration, consider a car's state defined by its position and heading direction.

$$\mathbf{x}_k = [x_k, y_k, \psi_k]^T$$

In this framework, the variable x_k denotes the robot's position along the x-axis at time k, while y_k signifies its position along the y-axis. The term ψ_k represents the robot's bearing angle.

This setup characterizes a robot navigating an environment equipped with a sensor in terms of two distinct scenarios: motion and observation. The "motion" scenario captures the robot's positional shift from time $k-1$ to k. Conversely, "observation" refers to the robot's detection of a landmark $\mathbf{m}_k^i$ at time k observed from position x_k. Focusing on motion, robots typically possess sensors like encoders or inertial sensors to gauge their movement. Such devices might register metrics like acceleration or angular velocity rather than directly recording positional changes. However, irrespective of the specific sensor employed, the general motion(process model) can be depicted as in eq.(4.4).

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}; \mathbf{u}_k; \mathbf{v}_k) \tag{4.4}$$

Let $\mathbf{u}_k$ represent the measurement from the motion sensor, often termed the control input, while $\mathbf{v_k}$ denotes the process noise. A general function f characterizes this process, though its specific operation remains undefined here. Such generality ensures that the function can encompass any motion sensor, thus granting universality to the equation. This function is also commonly labelled as the process model f in this study.

Complementary to this process model is an observation equation (or model). This equation outlines the scenario wherein the robot, situated at position $\mathbf{x}_k$, detects a landmark point $\mathbf{m}^i$.

$$\mathbf{M} = \mathbf{m}^1, \mathbf{m}^2, \mathbf{m}^3, ..., \mathbf{m}^i$$

, and each landmark maintains its own state, such as its position:

$$\mathbf{m}^i = [x^i, y^i]^T$$

an observation data $\mathbf{z}_k^{m^i}$ is generated. Similarly,

$$\mathbf{z}_k^{m^i} = h(\mathbf{m}^i; \mathbf{x}_k; \mathbf{w}_k) \tag{4.5}$$

where $\mathbf{w}_k$ represents as the observation noise, and i is the index of the landmark.

4.2 Bayesian networks

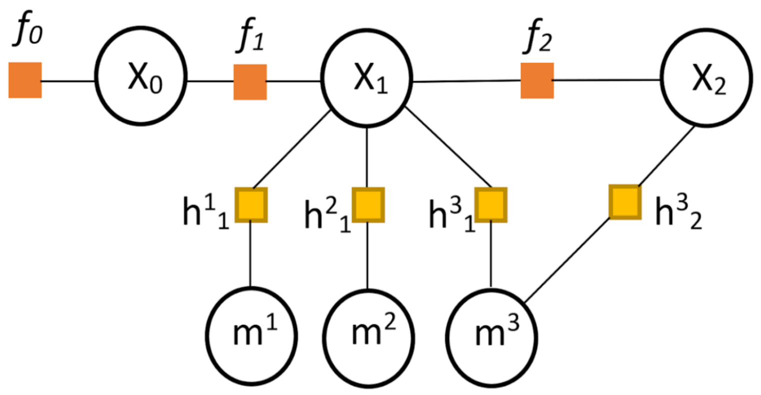

Figure 4.1: A simple SLAM example with three states and three landmarks. The orange boxes indicate the robot's motion with arrows, while the yellow boxes indicate distance and bearing measurement

Figure.4.1 provides a graphical representation of a basic SLAM problem. Here, a robot occupies three sequential poses: $\mathbf{x}_0$, $\mathbf{x}_1$, and $\mathbf{x}_2$. The system incorporates one prior information, f_0, and two process models, f_1 and f_2. The pose $\mathbf{x}_1$ is associated with three landmarks: $\mathbf{m}^1$, $\mathbf{m}^2$, and $\mathbf{m}^3$ through measurement models h_1^1, h_1^2, and h_1^3 respectively. Meanwhile, the $\mathbf{x}_2$ pose is connected to the landmark $\mathbf{m}^3$ via a measurement model, h_2^3. In the measurement model notations, the upper right index represents the landmark's identifier, while the lower right indicates the specific landmark observed from a particular pose.

In the context of the SLAM problem, the objective is to determine the robot's position utilizing data from its sensors. However, due to inherent uncertainties in measurements, obtaining an exact representation of the world becomes unfeasible. Instead, a probabilistic model can be developed to interpret and infer from the collected data. Eq.(4.6) aims to define the understanding of the unknown variables $\mathbf{S}$, which pertain to both the robot's poses and the undetermined landmark positions, based on a set of observed measurements $\mathbf{Z}$. In terms of Bayesian probability theory, this is captured by the conditional density, a process known as probabilistic inference [5].

$$p(\mathbf{S} \mid \mathbf{Z}) \tag{4.6}$$

where $\mathbf{S} = [\mathbf{X}, \mathbf{M}]^T$, and $\mathbf{Z}$ is measurement. For probabilistic inference, the Bayesian network, often referred to as the Bayes net, stands as a prevalent approach for modelling such problems within robotics.

According to [5], this directed graphical model features nodes symbolizing variables, denoted as θ_j. The complete set of relevant random variables is represented as $\Theta = \theta_1 ... \theta_n$. Within a Bayes net, the joint probability density $p(\Theta)$ across all variables Θ is expressed as the product of the conditional densities corresponding to each individual node:

$$p(\Theta) \triangleq \prod_j p(\theta_j | \pi_j) \tag{4.7}$$

In the provided eq.(4.7), $p(\theta_j|\pi_j)$ represents the conditional density associated with the node θ_j, while π_j denotes the value assignments for the parent nodes of θ_j [5].

For illustration, refer to the SLAM example in Figure.4.1. Here, the relevant random variables are denoted by $\Theta = \mathbf{S}, \mathbf{Z}$, where $\mathbf{S}$ stands for unknown poses and landmarks, and $\mathbf{Z}$ for measurements. The joint density, expressed as $p(\mathbf{S}, \mathbf{Z}) = p(\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{m}^1, \mathbf{m}^2, \mathbf{m}^3, \mathbf{z}_1^1, \mathbf{z}_1^2, \mathbf{z}_1^3, \mathbf{z}_2^3)$, is derived from the product of the given conditional densities:

$$\begin{aligned} p(\mathbf{S}, \mathbf{Z}) &= p(\mathbf{x}_0)p(\mathbf{x}_1 \mid \mathbf{x}_0)p(\mathbf{x}_2 \mid \mathbf{x}_1) && (4.8)\\ &\times p(\mathbf{m}^1)p(\mathbf{m}^2)p(\mathbf{m}^3) && (4.9)\\ &\times p(f_0 \mid \mathbf{x}_0) && (4.10)\\ &\times p(\mathbf{z}_1^1 \mid \mathbf{x}_1, \mathbf{m}^1)p(\mathbf{z}_1^2 \mid \mathbf{x}_1, \mathbf{m}^2)p(\mathbf{z}_1^3 \mid \mathbf{x}_1, \mathbf{m}^3)p(\mathbf{z}_2^3 \mid \mathbf{x}_2, \mathbf{m}^3). && (4.11) \end{aligned}$$

The joint density in this analysis can be categorized into four distinct sets of factors [5]:

- A Markov chain defined by $p(\mathbf{x}_0)p(\mathbf{x}_1|\mathbf{x}_0)p(\mathbf{x}_2|\mathbf{x}_1)$ for the poses $\mathbf{x}_0$, $\mathbf{x}_1$, and $\mathbf{x}_2$ as given in eq.(4.8). The conditional densities, $p(\mathbf{x}_{k+1} \mid \mathbf{x}_k)$, can be understood as containing prior knowledge or may be derived from pre-existing control inputs.
- Prior densities for the landmarks, specifically $p(\mathbf{m}^1)$, $p(\mathbf{m}^2)$, and $p(\mathbf{m}^3)$ as shown in eq. (4.9).
- A conditional density, $p(f_0|\mathbf{x}_0)$, associated with the absolute pose measurement of the initial pose $\mathbf{x}_0$ as described in eq. (4.10).
- A compounded set of conditional densities, $p(\mathbf{z}_1^1 \mid \mathbf{x}_1, \mathbf{m}^1)p(\mathbf{z}_1^2 \mid \mathbf{x}_1, \mathbf{m}^2)p(\mathbf{z}_1^3 \mid \mathbf{x}_1, \mathbf{m}^3)p(\mathbf{z}_2^3 \mid \mathbf{x}_2, \mathbf{m}^3)$, related to the four measurements from the landmarks $\mathbf{m^1}$, $\mathbf{m^2}$, and $\mathbf{m^3}$ as observed from the poses $\mathbf{x_0}$, $\mathbf{x_1}$, outlined in eq. (4.11).

The specific structure of the densities discussed is largely dictated by the application in question and the sensors in use. Commonly, these densities are represented using the multivariate Gaussian distribution [5], which has a probability density given by:

$$N(\theta;\mu,\Sigma) = \frac{1}{\sqrt{2\pi|\Sigma|}} \exp\left\{-\frac{1}{2}\|\theta-\mu\|_{\Sigma}^{2}\right\} \tag{4.12}$$

where $\mu \in \mathbb{R}^n$ is the mean, Σ is an $n \times n$ covariance matrix, and

$$\|\theta-\mu\|_{\Sigma}^{2} = (\theta-\mu)^T\Sigma^{-1}(\theta-\mu) \tag{4.13}$$

Often, Gaussian densities are utilized to specify priors on undetermined variables, and it is frequently both valid and practical to conceptualize measurements as being tainted by Gaussian noise with a zero mean. For instance, the association between a robot's pose and observed landmarks can be articulated as shown in eq.(4.14):

$$\mathbf{z}_k^{\mathbf{m^i}} = \begin{bmatrix} r_k^i \\ \beta_k^i \end{bmatrix} + \mathbf{w}_k^m \tag{4.14}$$

Where r_k^i is the range of distance between the detected object and the robot, and β_k^i is the bearing angle between them. In this context, the noise $\mathbf{w}_k^m$ follows a zero-mean Gaussian distribution characterized by the measurement covariance $\mathbf{R}$. This leads to the derived probabilistic measurement model $p(\mathbf{Z}|\mathbf{S})$, expressed as:

$$p(\mathbf{Z}|\mathbf{X},\mathbf{M}) = \frac{1}{\sqrt{|2\pi\mathbf{R}|}} \exp\left\{-\frac{1}{2}\left\|(\mathbf{w}_k^m)^T\mathbf{R}^{-1}(\mathbf{w}_k^m)\right\|_{\mathbf{R}}^{2}\right\} \tag{4.15}$$

and

$$\mathbf{w}_k^m = \begin{bmatrix} r_i^k - z_m^{i,1} \\ \beta_k^i - z_m^{i,2} \end{bmatrix} \tag{4.16}$$

However, it's crucial to highlight that Gaussian measurement noise won't always be the default assumption. It's also worth noting that not all probability densities are rooted in measurements. As depicted in Figure.4.1, there are densities like $p(\mathbf{x_{k+1}}|\mathbf{x_k})$, which also define a probabilistic motion model that the robot is anticipated to adhere to. This model might be derived from odometry readings, and if so, the approach mirrors the one outlined earlier. Alternatively, this motion model could emerge from known control signals $\mathbf{u}_t$. Typically, the conditional Gaussian assumption is employed, as represented by:

$$p(\mathbf{x_{k+1}}|\mathbf{x_k},\mathbf{u_t}) = \frac{1}{\sqrt{|2\pi\mathbf{Q}|}} \exp\left\{-\frac{1}{2}\left\|(\mathbf{v}_k)^T\mathbf{Q}^{-1}(\mathbf{v}_k)\right\|_{Q}^{2}\right\} \tag{4.17}$$

$\mathbf{v}_k$ serves as a process noise. Given that the focus is on a 2D plane, there are three dimensions to consider: x, y, and the bearing ψ. Accordingly, the covariance matrix $\mathbf{Q}$ is a 3×3 matrix.

4.3 Maximum a Posteriori Inference

In response to the impact of noise, the objective is to determine the pose $\mathbf{X}$ and the landmark's position ($\mathbf{M}$) on the map, along with their respective probability distributions, using noisy data $\mathbf{Z}$ and control input $\mathbf{u}$. This constitutes a state estimation challenge. Historically, given the sequential nature of data acquisition in SLAM, researchers heavily relied on filters, most notably the Extended Kalman Filter (EKF) and Particle Filter.

While the Kalman filter primarily emphasizes the estimation of the current state $\mathbf{x_k}$ and largely overlooks prior states, recent advancements lean towards non-linear optimization techniques. As detailed in Sec.2.2.2, these methods harness data from all time instants for state estimation, a strategy commonly termed 'smoothing'. Evidence suggests that these techniques outperform traditional filters [38], and they have since emerged as the prevailing methods in modern SLAM studies. Consequently, the most frequently utilized estimator for the unidentified state variables $\mathbf{X}$ is the Maximum a posteriori (MAP) estimate. This estimate maximizes the posterior density $p(\mathbf{X}|\mathbf{Z})$ of the states $\mathbf{X}$ given the measurements $\mathbf{Z}$, as represented in the eq.(4.18) below[5]:

$$\mathbf{X}_{\text{MAP}} = \arg\max_{X} p(\mathbf{X}|\mathbf{Z}) \tag{4.18}$$

$$= \arg\max_{X} \frac{p(\mathbf{Z}|\mathbf{X})p(\mathbf{X})}{p(\mathbf{Z})} \propto P(\mathbf{Z} \mid \mathbf{X})P(\mathbf{X}) \tag{4.19}$$

The equation (4.18) represents Bayes' theorem, detailing the posterior $p(\mathbf{X}|\mathbf{Z})$ as the multiplication of the measurement likelihood $p(\mathbf{Z}|\mathbf{X})$ with the prior $p(\mathbf{X})$ over states, all appropriately scaled by the normalization factor $p(\mathbf{Z})$. While directly computing the entire posterior distribution can be challenging, it is feasible to obtain an optimal state estimate by MAP estimation. In eq.(4.19), the denominator of Bayes' theorem is independent of the state $\mathbf{X}$, allowing it to be disregarded. Bayes net suggests that optimizing the posterior probability equates to maximizing the product of the likelihood and the prior:

$$\mathbf{X}_{\text{MAP}} = \arg\max_{\mathbf{X}} L(\mathbf{X};\mathbf{Z})p(\mathbf{X}). \tag{4.20}$$

Let $L(\mathbf{X};\mathbf{Z})$ represent the likelihood of observing the states $\mathbf{X}$ when given measurements $\mathbf{Z}$, referring to the eq.(4.16) for the likelihood function. This function is defined to be directly proportional to $p(\mathbf{Z}|\mathbf{X})$:

$$L(\mathbf{X};\mathbf{Z}) \propto p(\mathbf{Z}|\mathbf{X}) = f_{\mathbf{w}}\left(\mathbf{w}_k = \mathbf{l}\left[\mathbf{x}_k, \mathbf{z}_k\right]\right) \tag{4.21}$$

From the measurement model h,

$$\mathbf{l}\left[\mathbf{x}_k, \mathbf{z}_k\right] = \mathbf{z}_k - \mathbf{h}(\mathbf{m^i}; \mathbf{x}_k) \tag{4.22}$$

However, in the context of [5], It is claimed that despite the use of a linear measurement function, the dimensionality of the measurement variable $\mathbf{Z}$ often fails to fully capture the complexity of the underlying unknown variables it is associated with. The observed dimension differences lead to a probability density function of the unknown variables that is highly concentrated and lacks comprehensive constraining information. In order to obtain a comprehensive probability density and effectively limit the uncertainties associated with these variables, it is crucial to integrate data from various measurements. To address this difficulty, the following part will explore the implementation of the factor graph.

4.4 Factor graph

In the realm of modelling, filters like the Extended Kalman Filter (EKF), Particle Filter, and Bayesian Filter are widely recognized for their robust framework. However, for inference tasks, the graphical model is often preferred. This preference arises because of its remarkable extensibility and its capability to be applied recursively over k timesteps, as illustrated in eq.(4.23).

$$f\left(\mathbf{x}_{0:k} \mid \mathbf{z}_{0:k}, \mathbf{u}_{0:k}, \mathbf{x}_0\right) \propto f\left(\mathbf{x}_0\right) \prod_{i=1}^{k} f\left(\mathbf{x}_i \mid \mathbf{x}_{i-1}, \mathbf{u}_i\right) \times \prod_{i \in Z} L\left(\mathbf{x}_i; \mathbf{z}_i\right) \tag{4.23}$$

To motivate this, consider performing MAP inference for the SLAM example (Figure.4.1). After conditioning on the observed measurements $\mathbf{Z}$, the posterior $p(\mathbf{X}|\mathbf{Z})$ can be expressed as eq.(4.24), which re-written using eq.(4.19)

$$\begin{aligned} p(\mathbf{X}|\mathbf{Z}) \propto\, & p(\mathbf{x_0})p(\mathbf{x_1}|\mathbf{x_0})p(\mathbf{x_2}|\mathbf{x_1}) \\ & \times p(\mathbf{m^1})p(\mathbf{m^2})p(\mathbf{m^3}) \\ & \times L(\mathbf{x_0}; f_0) \\ & \times L(\mathbf{x_1}, \mathbf{m^1}; \mathbf{z_1^1})L(\mathbf{x_1}, \mathbf{m^2}; \mathbf{z_1^2})L(\mathbf{x_1}, \mathbf{m^3}; \mathbf{z_1^3})L(\mathbf{x_2}, \mathbf{m^3}; \mathbf{z_2^3}). \end{aligned} \tag{4.24}$$

The graphical models eq.(4.23) introduced are examples of Bayes networks, and these networks consist of multiplying terms together, which can be equivalently represented using factor graphs. A factor graph is a bipartite network with two separate types of nodes, namely factors and variables. As depicted in Figure.4.1, the pose state $\mathbf{x_k}$ and landmark $\mathbf{m^i}$ are denoted as variables. Conversely, the boxes represented in the graph are factors, with edges connecting factor nodes and variable nodes. It's crucial to differentiate this from Bayesian networks: in the context of SLAM, the variables represent components to be optimized, such as the robot's pose $\mathbf{X}$ and landmarks $\mathbf{M}$, excluding observations $\mathbf{Z}$ and inputs $\mathbf{u}$. This is because the latter quantities are given, not subject to optimization. The factor nodes show the relationships between these optimizable variables, originating from motion and observation equations. A global factor graph equation can be defined as follows,

$$\boldsymbol{g}(\mathbf{S}) = \prod_i \boldsymbol{g}_i(\chi_j) \tag{4.25}$$

Let $\mathbf{S} = \mathbf{x_0}, ..., \mathbf{x_k}, \mathbf{m^1}, ..., \mathbf{m^i}$ represent the set of variable nodes. Additionally, for each j, χ_j denotes the subset of $\mathbf{S}$ comprising adjacent variable nodes. Applying this formulation (4.25) to the SLAM example presented in Figure.4.1, the resulting factor graph factorization is as follows:

$$\begin{aligned} \boldsymbol{g}(\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \mathbf{m^1}, \mathbf{m^2}, \mathbf{m^3}) = \, & \boldsymbol{g}_1(\mathbf{x_0})\boldsymbol{g}_2(\mathbf{x_1}, \mathbf{x_0})\boldsymbol{g}_3(\mathbf{x_2}, \mathbf{x_1}) \\ & \times \boldsymbol{g}_4(\mathbf{m^1})\boldsymbol{g}_5(\mathbf{m^2})\boldsymbol{g}_6(\mathbf{m^3}) \\ & \times \boldsymbol{g}_7(\mathbf{x_0}) \\ & \times \boldsymbol{g}_8(\mathbf{x_1}, \mathbf{m^1})\boldsymbol{g}_9(\mathbf{x_1}, \mathbf{m^2})\boldsymbol{g}_{10}(\mathbf{x_1}, \mathbf{m^3})\boldsymbol{g}_{11}(\mathbf{x_2}, \mathbf{m^3}). \end{aligned} \tag{4.26}$$

Chapter 5

Methodology

In this chapter, the primary focus is on incorporating the learned robot motion and observation with the factor graph for state estimation. Through this, the dynamic motion and observation are trained via supervised learning and then incorporated back into the factor graph. Within the realm of deep learning, supervised learning technique is trained using labeled data. This means that during the training process, the model is provided with input-output pairs. It learns to map the input to the corresponding output based on these pairs. Factor graph, known as bipartite graphs, characterized by their ability to manage conditional independence and efficiently handle probabilistic densities, have become integral in solving numerous real-world challenges in robotics. In the realm of state estimation, this study introduces a framework where the estimated quantities, denoted as $\mathbf{x}$, are treated as variable nodes. Information derived from dynamic models and sensor inputs is represented as factor nodes. Under the assumption of Gaussian noise, each factor encodes a joint probability density for specific values of $\mathbf{x}$. By using maximum a posteriori (MAP) inference on a factor graph to solve the nonlinear least squares optimization, and this approach involves mathematically expressed through a function that incorporates the Mahalanobis norm of an error vector, factoring in covariance and other relevant parameters. The methodology also presents an exploration of this optimization process, involving techniques such as the Gauss-Newton and Levenberg-Marquardt algorithms to solve a sequence of linear approximations of the error (cost) function.

5.1 State Estimation in Factor Graph Structure

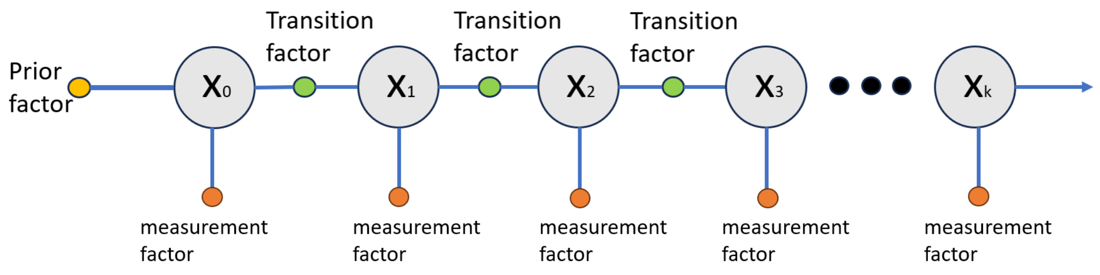

Figure 5.1: Factor graph representation of the learned smoothers

Factor graphs represent a category of probabilistic graphical models, and each factor signifies a constraint or measurement associated with the variables of the problem, such as robot poses and landmark positions. For example, in Figure. 5.1, the transition factors are constraints originating from the dynamic model between consecutive robot poses, while measurement factors are constraints stemming from sensor measurements that link robot poses to landmark positions. When using factor graphs to estimate a state, an optimizer is used to get the total error as low as possible across all factors. This solves the error function, which is defined by the sum of the costs or errors for each factor. The error function, typically framed as a nonlinear least squares problem, is central to adjusting the variables within the graph—such as poses and landmarks—to adhere to the constraints and measurements depicted. The optimization process, which may utilize algorithms like Levenberg-Marquardt, Gauss-Newton, or ISAM2, seeks to identify the configuration of variables that minimizes the overall cost, effectively enhancing the accuracy and reliability of the state estimation. The utilisation of direct optimisation, as indicated by equation (4.23), carries the potential drawback of numerical overflows or underflows. Significantly, this function is observed to be a result of non-negative factors. Therefore, it is reasonable to utilise the logarithm function in order to mitigate this issue.

$$\mathbf{x}^* = \arg\max_{X} f\left(\mathbf{x}_{0:k} \mid \mathbf{z}_{0:k}, \mathbf{u}_{0:k}, \mathbf{x}_0\right) \tag{5.1}$$

Substituting, the definition of a factor graph is that

$$\ln p(\mathbf{x_{1:k}} \mid \mathbb{I}_k) = \sum_{i=1}^{k} \ln f(\mathbf{x_k} \mid \mathbf{x_{k-1}}, \mathbf{u_k}; \theta_n) + \sum_{i=1}^{k} \ln h(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_m) + \ln f_0(\mathbf{x_0}) + C \tag{5.2}$$

Where $\mathbb{I}_k = \{\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0\}$, the state transition probability is represented by $f(\mathbf{x_k} \mid \mathbf{x_{k-1}}, \mathbf{u_k}; \theta_n)$, which is parameterized by θ_n. Similarly, the observation likelihood is denoted by $h(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_m)$ and is parameterized by θ_m. However, in this project, certain assumptions are operated under. All models are relative to one another; therefore, the distribution is only over the difference between the start and end states of the robot and does not depend upon the absolute value of the start and end states. For the process model,

$$f(\mathbf{x_k} \mid \mathbf{x_{k-1}}, \mathbf{u_k}; \theta_n) = f(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f) \tag{5.3}$$

It is important to note that the operator $\ominus$ signifies the manifold difference, managing complexities such as angle wrapping. Correspondingly, a parallel expression is in the context of the observation model, wherein the relative displacement is computed.

$$h(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h) = h(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h) \tag{5.4}$$

Under the presumption of Gaussian error (eq. 4.17), the logarithm of the process model distribution assumes the following form:

$$\ln f(\mathbf{x_k}, \mathbf{x_{k-1}}, \mathbf{u_k}; \theta_n) = -\frac{1}{2} e(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f)^\top \mathbf{\Omega}_Q(\mathbf{u}_k; \theta_n) e(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f) + D \tag{5.5}$$

The function $e(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f)$ is the error function, which is also the factor constraint for each variable $\mathbf{x}$ (vertices), and this method computes the difference between the received and estimated measurement for the vehicle's relative transformation. Its error is given by

$$e(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f) = (\mathbf{x_k} \ominus \mathbf{x_{k-1}}) \ominus n(\mathbf{u_k}; \theta_n) \tag{5.6}$$

The term $(\mathbf{x_k} \ominus \mathbf{x_{k-1}})$ is the relative transformation between the two configurations of the robot computed using the current estimate received in the graph. The term $n(\mathbf{u_k}; \theta_n)$ is the prediction of the relative transformation from the learned process model. The term $\mathbf{\Omega}_Q(\mathbf{u}_k; \theta_n)$ is the information matrix for the process model. Similarly, the measurement model distribution (eq. 4.15) assumes the following form:

$$\ln h(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h) = -\frac{1}{2} e(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h)^\top \mathbf{\Omega}_R(\mathbf{x_k}, \mathbf{u_k}; \theta_n) e(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h) + F \tag{5.7}$$

The function $e(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h)$ is the error function for the observation model. Its value is given by

$$e_k(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h) = \mathbf{z_k} \ominus n(\mathbf{x_k}, \mathbf{u_k}; \theta_h) \tag{5.8}$$

The term $\mathbf{z_k}$ is the distance between the robot and the landmark computed using the current estimate in the factor graph. The term $n(\mathbf{x_k}, \mathbf{u_k}; \theta_h)$ is the prediction of the relative displacement from the learned observation model. The term $\mathbf{\Omega}_R$ is the information matrix for the observation model. In summary, the network needs to learn (for prediction) $n(\mathbf{u_k}; \theta_n)$, $n(\mathbf{x_k}, \mathbf{u_k}; \theta_h)$, $\mathbf{\Omega}_Q$, and $\mathbf{\Omega}_R$.

In estimating the unknown variable that the robot poses in the trajectory and the observation, the Gaussian distribution densities defined by each factor play a pivotal role. By employing Maximum A Posteriori (MAP) inference on the factor graph, the robot's position can be determined. Given the assumption of Gaussian noise, this estimation challenge can be translated into a nonlinear least squares optimization task. Following from eq.5.1, the solution presents numerous negative values. For ease of computation and interpretation, it may be beneficial to adjust the scale and reverse these signs.

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} = 2 \ln f'(x) \tag{5.9}$$

Therefore, the goal is to compute

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} c\left(\mathbf{x_{1:k}} \mid \mathbb{I}\right) \tag{5.10}$$

where,

$$\begin{aligned} c\left(\mathbf{x}_{1:k} \mid \mathbb{I}_k\right) = \sum_{i=1}^{k} & e(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f)^\top \mathbf{Q}_k^{-1}(\mathbf{u_k}; \theta_n) e(\mathbf{x_k} \ominus \mathbf{x_{k-1}} \mid \mathbf{u_k}; \theta_f) \\ & + e(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h)^\top \mathbf{R}_k^{-1}(\mathbf{x_k}, \mathbf{u_k}; \theta_n) e(\mathbf{z_k} \mid \mathbf{x_k}, \mathbf{u_k}; \theta_h) \end{aligned} \tag{5.11}$$

5.2 Learning Robot Motion

5.2.1 Introduction to robot motion modeling

The fundamental idea underlying our work is that the model will acquire estimation of the robot's motion $n(\mathbf{u_k}; \theta_n)$, measurement $n(\mathbf{x_k}, \mathbf{u_k}; \theta_n)$, and the information matrix $\Omega_Q(\mathbf{u_k}, \theta_n)$, $\Omega_R(\mathbf{x_k}, \mathbf{u_k}, \theta_n)$ by utilising control input (wheel speed) and prior information for factor graph, as referenced in equation (5.6) and (5.8). This means the neural network provides the expected transformation or change in the robot's state according to the process model. Similarly, the landmark detection is based on the observation deriving from the control input and the location of the robot. Comprehending how the robot is anticipated to move or transition between states in reaction to inputs and prior states is crucial. The rationale for not learning the absolute position, but rather the relative displacement between each robot's state, is that modelling a robot's actions based on absolute positions would require learning how to handle the robot's behaviour at every possible position in the space, which is impractical. This task is extremely complex due to the robot's potential for being in numerous positions. In short, the robot's movement is shift-invariant, indicating that it behaves consistently regardless of its exact position in space, as long as the control input remains the same. For instance, the way in which a robot rotates or travels does not fundamentally differ based on its location on a map when the control input is the same, be it the top left corner or the bottom right. The model represents this shift-invariant by emphasising relative changes. The robot's learning process focuses on understanding its behaviour during transitions, such as turning and accelerating, rather than solely at specific spots. Furthermore, models that rely on relative transformation exhibit greater generalisation. If a robot is taught in a certain region (such as a rectangle) and subsequently relocated to a different region, a model relying on absolute positions may be inadequate due to its lack of familiarity with the new positions. This new space does not have the distribution of it. Nevertheless, a model that relies on relative transformations has the ability to adapt more easily to unfamiliar surroundings. As it is not concerned with the precise location of the robot, but rather its movement or change in state resulting from control input, this approach remains effective even in unfamiliar or unexplored areas.

5.2.2 Loss function

The choice to employ supervised learning stems from the given ground truth with noise feature, characterized by a Gauss-distribution zero mean and diagonal covariance matrix. As our aim is to derive the robot motion directly from given trajectory, supervised learning is ideal. In this supervised neural network learning methodology, we seek to minimize the loss function by Mean Squared Error (MSE). To obtain the relative transformation information, the loss function is defined as:

$$L_{\Delta_f} = \sum_k \left((\Delta\overline{x_k} - \Delta x_k^*)^2 + (\Delta\overline{y_k} - \Delta y_k^*)^2 + (\Delta\overline{\phi_k} - \Delta\phi_k^*)^2\right) \tag{5.12}$$

where $\Delta\overline{x_k}$, $\Delta\overline{y_k}$, $\Delta\overline{\phi_k}$ and Δx_k^*, Δy_k^*, $\Delta\phi_k^*$ are the estimated and true robot transformation for time k, respectively.

$$L_v = \sum_k \left((\overline{v_k}^x - v_k^{x*})^2 + (\overline{v_k}^y - y_k^{y*})^2 + (\overline{v_k}^\phi - v_k^{\phi*})^2\right) \tag{5.13}$$

Similarly, for the noise of the process model, where $\overline{v_k}^x$, $\overline{v_k}^y$, $\overline{v_k}^\phi$ and v_k^{x*}, y_k^{y*}, $v_k^{\phi*}$ are the estimated and true robot process noise for time k, respectively. On the other hand, the measurement between the landmark and the robot's loss is presented as:

$$L_{\Delta_h} = \sum_k \left((\Delta\overline{r_k} - \Delta r_k^*)^2 + (\Delta\overline{\psi} - \Delta\psi_k^*)^2\right) \tag{5.14}$$

Where $\Delta\overline{r_k}$ $\Delta\overline{\psi}$ and Δr_k^*, $\Delta\psi_k^*$ are the estimated and true range for time k, respectively. Additionally, the prediction of measurement noise compared with the label,

$$L_w = \sum_k \left((\overline{w_k}^r - w_k^{r*})^2 + (\overline{w}^\psi - w_k^{\psi*})^2 \right) \tag{5.15}$$

Chapter 6

Implementation & Experiments

In this chapter, an in-depth study of tools for learning robot motion and observation with factor graph is presented. Libraries such as **GTSAM** and **TensorFlow** to bolster computational effectiveness and refine experimental approaches. The experimental setup is detailed, commencing with the generation of robot trajectory. During this phase, the robot's trajectory is influenced by Gaussian noise, highlighting the challenges posed by real-world conditions. The discussion then transitions to the training stage, where the **TensorFlow** neural network was chosen for this project. This should include considerations of the model's architecture, its suitability for our data, and any other parameter tuning that influenced the decision. Furthermore, analyze how the model performed during the training phase and on the test dataset. After obtaining the prediction motion and observation model, the **GTSAM** library was utilised for building SLAM system where the factor graph was integrated with the estimation from the neural network.

6.1 Library Requirement

During this research, we integrated two critical libraries to enhance computational efficiency and streamline experimental processes:

1. **GTSAM**: Georgia Tech Smoothing And Mapping is an open-source C++ toolkit designed for robotic mapping, state estimation, and tackling universal problems involving nonlinear optimization. The basic idea underlying GTSAM is to provide a powerful and easy platform that enables the building and optimization of factor graphs for diverse applications, including Simultaneous Localization and Mapping (SLAM). In our case, we adapt the Python wrappers of GTSAM for building the factor graph.

2. **TensorFlow**: Developed by Google, TensorFlow is a widely recognized open-source machine learning framework. Within the scope of this project, it was utilized to learn the robot motion and observation. TensorFlow's extensive tools and features, combined with its high-performance computational backend, render it perfect for managing intricate neural network designs and processing large datasets.

6.2 Assumptions and Noise Model Setup

For this research, we always assume that we have access to the ground truth trajectory of the robot state $\mathbf{x}_{t=0...T}$. To gather training samples within the simulation environment with the purpose of predicting the motion and observation of the robot, the robot is assumed to be simulated in a 2D map, where the robot's position is determined by three parameters: x, y coordinate, and the bearing angle. Based on these training samples, the noise model is conventionally perceived as uncorrelated Gaussian noise, characterized by a zero mean and a 3×3 constant diagonal covariance matrix.

$$n_t \sim \mathcal{N}(0, \Sigma) \tag{6.1}$$

where

$$\mathbf{Q} = \begin{bmatrix} q_x & 0 & 0 \\ 0 & q_y & 0 \\ 0 & 0 & q_\theta \end{bmatrix} \tag{6.2}$$

The variance can vary because of differences in the units used—specifically, one state is measured in radians, which typically have numerically smaller uncertainty compared to the x and y coordinate positions. Consequently, in this study, we have set the variances q_x and q_y at 0.1, and q_θ at 0.01. Additionally, the fact that all off-diagonal elements are zero indicates that the noise in the x, y, and rotational ψ are independent of each other. On the other hand, observation noise is influenced by two variables: the distance between the landmark and the robot, and the bearing angle between them. Consequently, in this study, the observation noise model is assumed to be uncorrelated Gaussian noise, characterized by zero mean and a 2×2 constant diagonal covariance matrix.

$$\mathbf{R} = \begin{bmatrix} r_{range} & 0 \\ 0 & r_\theta \end{bmatrix} \tag{6.3}$$

Here, r_{range} is set at 0.1, and r_θ is set at 0.01. Nevertheless, this generalisation process and observation noise model may not consistently correspond with real-world systems. A considerable number of the systems can be more precisely characterised by heteroscedastic noise models, where the level of uncertainty is dependent on the state of the system or the prospective control inputs [1, 38]. The inclusion and acknowledgement of heteroscedasticity in system dynamics has been experimentally shown to improve filtering effectiveness in many robotic applications [1, 38]. However, for the sake of clarity and ease in the foundational stages of this project, it will maintain the assumption of the noise model in the simulation trajectory is uncorrelated Gaussian with a zero mean and diagonal covariance matrix.

6.3 Trajectory simulation

The primary objective of the first-stage experiment was to simulate the robot's trajectory with a 500 number of steps that have been used to learn the robot's motion from the control input, wheel speed. In this study, the map was designed as a two-dimensional map with 500 fixed landmarks (Figure.6.1). The x-coordinate, the y-coordinate, and the orientation angles are the only three factors in the map that affect the robot's position. These parameters are crucial for accurately depicting the robot's location within the mapped environment, ensuring precise navigation and interaction with the landmarks and other elements of the map.

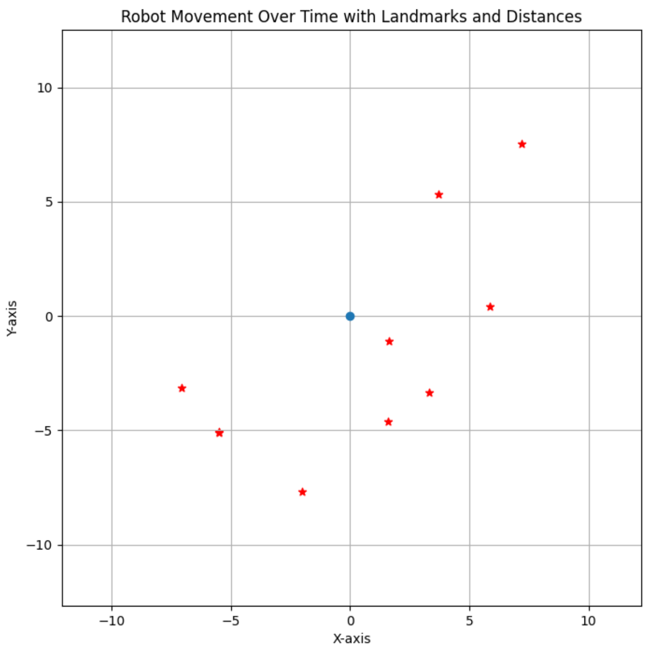

Figure 6.1: 2D map displaying nine landmarks (red dots), and the initial state of the robot (blue dot) at the origin $(0, 0)$, oriented at 0 degrees relative to the global frame.

In the trajectory simulation (Left Figure.6.2), the training data comprises a diverse set of trajectories, each generated randomly and incorporated with Gaussian distribution noise. For each trajectory, two critical parameters are sampled randomly: the wheel speed and the turning angle, both chosen from predefined. The specific range for wheel speed for x and y direction is between 0 and 2 meters per second. Similar to angular velocity, the range for it is defined between 0 and 0.2 radians per second. As for the range of the turning angle, it was set to a full range of 0 to 2π, which covers all possible angles in radians. Each trajectory is designed to be 500 steps in length, with every step labeled with the true robot pose. Alongside these steps, the simulated sensor detection is equipped on the robot to detect a circular range with a radius of 5 meters, using itself as the center, as illustrated in the right Figure.6.2

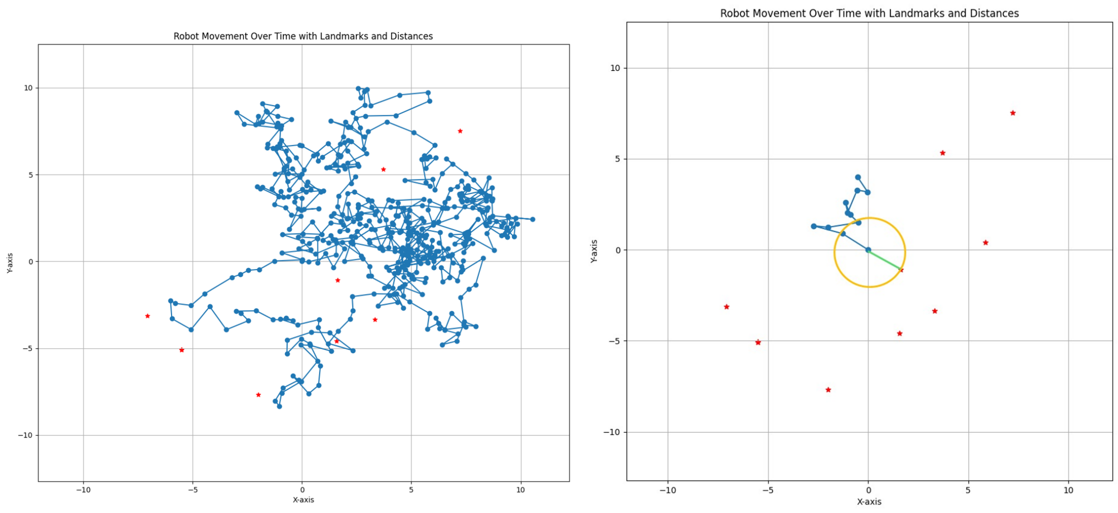

Figure 6.2: Left: Robot trajectory with random wheel speed and steering angle, Right: Sensor detection with a radius of 5 meters (yellow circle)

The illustration in Figure 6.2 presents the trajectory of the robot under the influence of Gaussian distribution noise, a scenario that closely mirrors real-world conditions. In such environments, a variety of factors contribute to uncertainties in the robot's path. These can include, but are not limited to, sensor errors and limitations inherent in the dynamic models used. Specifically, in our study, we assume the primary source of noise is ground friction. This friction affects the robot's tyres, leading to variations in wheel speed and, consequently, trajectory uncertainty. In addition, as illustrated on the right side of Figure. 6.2, the green line connecting the robot's location (indicated by a blue dot) and the landmark (marked by a red dot) represents the measurement detected by the robot's sensor. This measurement is not only assumed to be a direct reading from the sensor but also includes an element of noise. This noise is attributed to the inherent uncertainty of the sensor's performance. In other words, while the sensor aims to accurately measure the distance and direction to the landmark, various factors, such as environmental conditions, sensor quality, and operational limitations, can introduce some degree of error or variability into these measurements. Consequently, the green line, while representing the detected measurement, also encapsulates the uncertainty associated with the sensor's ability to precisely gauge the distance and angle to the landmark.

6.3.1 Landmark Generation

In the context of SLAM, landmarks play an essential role as they serve as reference points for a robot or vehicle to determine its position within an unknown environment. The accuracy and distribution of these landmarks directly impact the effectiveness of the SLAM algorithm. In this study, to facilitate SLAM simulations or to test algorithms under controlled conditions, a predefined set of landmarks is used. These landmarks are assumed to be known a priori, representing the "ground truth" in simulation environments. In real-world applications, landmarks can be obtained through various means, such as GPS coordinates or through sensory data collected by equipped vehicles, for instance, using laser range-finders.

The provided Python script (Code Listing.6.1) is designed to simulate the process of generating a file (landmark.txt) containing a list of landmarks for use in SLAM simulations. The landmarks are represented as red points in a 2D space, with two coordinates uniformly distributed within the range of '-max_x' to 'max_x' and '-max_y' to 'max_y', respectively. Additionally, by seeding the random number generator ('np.random.seed(42)'), the script ensures that the generated landmarks are reproducible across different runs, which is crucial for testing and comparison purposes.

```python
def generate_landmarks_file(filename, num_landmarks, max_x, max_y):
    # Seed the random number generator for reproducibility
    np.random.seed(42)

    # Generate landmarks
    landmarks = np.random.uniform(low=-max_x, high=max_x, size=(
        num_landmarks, 2))

    # Write landmarks to file
    with open(filename, 'w') as f:
        for landmark in landmarks:
            f.write(f"{landmark[0]}, {landmark[1]}\n")

if __name__ == "__main__":
    # Parameters
    num_landmarks = 500
    max_x = 60
    max_y = 60
    filename = "landmarks.txt"

    # Generate the file
    generate_landmarks_file(filename, num_landmarks, max_x, max_y)
    print(f"Landmarks file '{filename}' generated successfully.")
```

Listing 6.1: Landmark generation

6.3.2 Dynamic model

The process model $f(.)$ for the prediction step in the robot trajectory simulation can be implemented using a known analytical model:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta T_k \mathbf{M}(\psi_k)(\mathbf{u}_k + \mathbf{v}_k) \tag{6.4}$$

In eq.(6.4), the term ΔT_k denotes the time interval between two successive robot states, which is always assumed to be 1 in our study. The matrix $\mathbf{M}(\psi_k)$ defines the rotation transformation from the vehicle-fixed frame to the world-fixed frame. Further, $\mathbf{u_k}$ indicates the control input, while $\mathbf{v_k}$ is a representation of the process noise. The relationships between these variables are expressed as follows:

$$M(\psi_k) = \begin{bmatrix} \cos\psi_k & -\sin\psi_k & 0 \\ \sin\psi_k & \cos\psi_k & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.5}$$

$$u_k = \begin{bmatrix} s_x \\ s_y \\ \dot{\psi}_k \end{bmatrix} \tag{6.6}$$

where s_x is the wheel speed along x axis of the vehicle and s_y is the speed along y axis of the vehicle; similarly, $\dot{\psi}_k$ denotes the angular velocity.

$$v_k = \begin{bmatrix} v_k \\ v_y \\ v_{\dot{\psi}} \end{bmatrix} \tag{6.7}$$

The process noise is assumed to have a zero mean, a Gaussian distribution, on all three dimensions of training samples. Given that ψ_k is non-Euclidean with $\psi_k \in [-\pi, \pi)$ and both $\mathbf{x_k}$ and $\mathbf{x_{k+1}}$ are members of the Special Euclidean group, utilizing $+$ as described in eq.(6.4) can result in angular discontinuities, also known as angle wrapping. To address this for computing the error (as illustrated at eq.(5.6)), the operation $\ominus$ as outlined in [12] is adopted.

$$\Delta\mathbf{x}_k = \mathbf{x}_{k+1} \ominus \mathbf{x}_k = \begin{bmatrix} \Delta T_k(\cos(\psi_k)\Delta x_k - \sin(\psi_k)\Delta y_k) \\ \Delta T_k(\sin(\psi_k)\Delta x_k + \cos(\psi_k)\Delta y_k) \\ \text{normAngle}(\Delta\psi_k) \end{bmatrix} \tag{6.8}$$

where relative transformation $\Delta\mathbf{x}_k = \Delta T_k \mathbf{M}(\psi_k)(\mathbf{u}_k + \mathbf{v}_k)$ is used to compute the error between the initial estimate and the prediction from the neural network.

The `dynamic_model` function (see Listing: 6.2) simulates the movement of a robot by updating its state based on control inputs and noise. It takes the current state $\mathbf{x}_k$, orientation ψ_k, control inputs $\mathbf{u}_k$, and process noise $\mathbf{v}_k$. Depending on the specified noise type (`gaussian` or `uniform`), it generates appropriate noise $\mathbf{v}_k$ using either a multivariate normal distribution or uniform distribution. The function calculates a new state by rotating the control inputs (accounting for the robot's orientation), adding the noise, scaling by the time interval Δt, and finally updating the robot's position. The orientation angle is normalized (Appendix code listing A.5) to keep it within the range $[-\pi, \pi]$ before returning the updated state.

```
def dynamic_model(x_k, psi_k, u_k, v_k):
    if dynamic_noise_type == 'gaussian':
        v_k = np.random.multivariate_normal([0, 0, 0], Q_std_dev)
    else: # uniform noise
        v_k = np.random.uniform(-Q_uniform_range, Q_uniform_range, size
            =(3,))

    M_psi_k = rotation_matrix(psi_k)
    a = M_psi_k @ (u_k.T + v_k.T)
    c = delta_t * a
    d = x_k + c
    d[2] = normalize_theta(d[2])
    return d
```

Listing 6.2: Dynamic model function.

6.3.3 Observation model

Alongside the process model, the robot's sensor detection also plays a vital role. Assuming the robot, equipped with a sensor that has a circular detection range around a radius of 5 meters and using itself as the center, can effectively monitor its surroundings within this area. This means that any object or landmark within a 5 meter radius of the robot can be detected and analyzed. In this study, we adopt a method that focuses on extracting only the nearest landmark to the robot at each of its poses. This approach means we will collect one measurement at each robot pose instead of several measurements simultaneously. Practically, the robot is capable of detecting several landmarks within the range of its sensor detection. However, for the ease of collecting training data at specific samples of interest, we initially assume that the robot can detect only one landmark at a time, specifically the one closest to it. Given this setup, eq.(4.14) would possibly describe the mathematical relationship that the robot uses to interpret sensor data. This eq.(6.9) and eq.(6.10) involves calculating the range of distance between detected objects relative to the robot as well as the bearing angle, which is used for computing the observation error followed by eq.(5.8).

$$r_k^i = \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2} + w_k^r \tag{6.9}$$

$$\beta_k^i = (atan2(y_i - y_k, x_i - x_k) - \psi + w_k^\beta) \tag{6.10}$$

The coordinates of the landmarks are denoted as x_i, y_i, while x_k, y_k represent the robot's position. Additionally, ψ signifies the orientation of the robot's pose in the global frame of reference, and w_k is the measurement noise in the sensor detection.

The `observation_model` (see Listing.6.3) function represents the observation process in a robot's sensor system. It accepts the coordinates (x_i, y_i) of a landmark and the current state (x_k, y_k, ϕ_k) of the robot, along with a noise vector w_k. Depending on the specified noise type (`Gaussian` or `uniform`), the function generates noise w_k using either a multivariate normal distribution or a uniform distribution. The function then calculates the observation model output: r_{k_i}, the noisy range measurement from the robot to the landmark, is computed as the Euclidean distance between the robot and the landmark with an added noise component $w_k[0]$. Similarly, β_{k_i}, the noisy bearing measurement, is computed as the arctangent of the relative position of the landmark with respect to the robot, offset by the robot's orientation ϕ_k and an additional noise component $w_k[1]$. The function returns these two measurements as an array.

```python
# Observation model function
def observation_model(x_i, y_i, x_k, y_k, phi_k, w_k):
    if observation_noise_type == 'gaussian':
        w_k = np.random.multivariate_normal([0, 0], R_std_dev)
    else:
        w_k = np.random.uniform(-R_uniform_range, R_uniform_range, size
            =(2,))

    r_k_i = np.sqrt((x_i - x_k)**2 + (y_i - y_k)**2)  + w_k[0]
    beta_k_i = np.arctan2((y_i - y_k), (x_i - x_k)) - phi_k + w_k[1]
    return np.array([r_k_i, beta_k_i])
```

Listing 6.3: Observation model function.

6.4 Data visualization

Appendix.A.1 creates a visual representation of a robot's movement in relation to landmarks using Matplotlib as shown in Figure.6.2. The script begins by setting up a figure and axes with `plt.subplots`, specifying a size of 8×8 inches. The plot's title is set to 'Robot Movement Over Time with Landmarks and Distances', and labels are assigned to the X and Y axes. Grid lines are enabled for better visibility. The robot's trajectory is plotted on these axes, with its initial position marked by a circle ('o') marker at $(0, 0, 0)$. The landmarks are scattered on the plot, represented by red asterisks. The X and Y limits of the axes are set based on the minimum and maximum coordinates of the landmarks, extended by a margin of 5 units for clarity. The canvas is then updated with `fig.canvas.draw()` to reflect these changes. Additionally, interactive zoom functionality is enabled by connecting a scroll event to the `on_scroll` function. This allows users to zoom in and out on the plot. The commented section of the code describes a possibility of a connection (a blue arrow) can be plotted from the robot to each landmark it detects, using the `ConnectionPatch` object. This feature would visually represent the interaction between the robot and the landmarks in its vicinity.

6.5 Data saving

The `save_to_csv` function is designed to write simulation data to a CSV file (Appendix code listing.A.2). It takes several parameters: the filename ('filename'), accumulated step count ('acc_step'), robot trajectory ('trajectory'), control inputs ('control_inputs'), relative poses ('relative_poses_list'), observations ('obs_list'), covariances ('covariances'), and a flag indicating whether to write the header ('write_header'). The function opens the file in write mode if 'write_header' is true, otherwise in append mode. If the header is to be written, it includes columns for step count, pose, control inputs, relative poses, observations, and covariance values. The function then iterates over the data, formatting covariance and observation values to four decimal places for precision. Each row in the CSV file corresponds to a single time step, containing the step number, current pose, control inputs, relative pose, observations, and formatted covariance values. This structured data storage is crucial for subsequent Deep Neural Networks training.

6.6 Deep Neural Network

The code snippet shown in Appendix.A.3, is a machine learning pipeline for training a neural network using data from a CSV file, generated previously. The process begins with loading the dataset using Pandas' `read_csv` function, specifying the path to the CSV file. The dataset contains columns representing control inputs, robot's pose, observations, relative transformations, and covariance.

6.6.1 Data collection

The data collection involves running the trajectory simulation at least 120 times, with each trajectory consisting of 500 steps. The initial training dataset, consisting of $60,000$ samples, with each sample having 16 features (6 inputs and 10 outputs, also known as labels), follows a Gaussian distribution. This process begins with loading the dataset using Pandas' `read_csv` function. The dataset has three inputs that denote the wheel speed in the x and y velocity, as well as the angular velocity. Additionally, there are three inputs that indicate the robot's poses, including the x and y coordinates and the orientation of the robot. On the other hand, there are two outputs: the bearing angle and the range, which represent the observation between the landmark and the robot. There are three outputs for predicting the relative transformation in the x and y directions, as well as the orientation direction. Moreover, the 5 outputs of noise for both process and observation noise. In summary, there are a total of 16 features in the dataset. The reason for choosing wheel speed features is to capture the essential spatial and movement characteristics of the robot, and the robot's pose determines the measurement between landmarks. Furthermore, an additional $12,000$ samples, generated from simulated non-Gaussian distribution trajectories, are used for the testing set. This approach not only ensures a thorough training process with a comprehensive validation phase but also enhances the robustness of the network by exposing it to varied data distributions, particularly non-Gaussian in the testing phase, to better prepare it for real-world scenarios. Additionally, the evaluation of the training result will not only include a non-Gaussian dataset but also generate a baseline testset with a Gaussian noise distribution for comparison.

6.6.2 Data preprocessing

The dataset comprises various features, including the robot's pose (x, y, theta), control inputs (u_k_x, u_k_y, u_k_theta), and the target variables y related to the robot's movement, observations, and covariance. In our methodology, normalization was implemented using the **StandardScaler** from Python's **scikit-learn** library. This scaler removes the mean and scales the data to unit variance. Specifically, each feature value is transformed using the formula:

$$\text{scaled\_value} = \frac{\text{original\_value} - \text{mean}}{\text{standard\_deviation}} \tag{6.11}$$

This transformation is applied to each feature independently. After normalization, all features roughly follow a Gaussian distribution with a mean of 0 and a standard deviation of 1, making them comparable in scale and more suitable for training the neural network. By normalizing the data in this manner, we ensured that the input features of our neural network model were on a comparable scale, avoiding different units and scales (e.g., pose coordinates in meters, angles in radians, and velocities in meters per second). Additionally, neural networks use gradient-based optimization techniques. When the input features are on similar scales, the gradient descent can converge faster, facilitating a more efficient and effective training process.

The dataset is then divided into 80% for training and 20% for validation, translating to $48,000$ samples for training and $12,000$ for validation. This split is important for training the model on a large dataset while also having a separate dataset to evaluate its performance during the training process.

6.6.3 Neural Network Architecture

A neural network model (Appendix Code listing.A.3) developed for this project is a fully connected feedforward network defined using TensorFlow and Keras, consisting of an input layer shaped according to the number of features, and multiple dense layers with ReLU activation functions. The output layer has dimensions corresponding to the target variables. Moreover, the first and second hidden layers each consist of 64 neurons, and the third hidden layer has 32 neurons. All hidden layers use the ReLU (Rectified Linear Unit) activation function. The ReLU function is chosen for its efficiency and effectiveness in non-linear transformations. The choice of having three hidden layers with these specific neuron counts is aimed at creating a model complex enough to capture the underlying patterns in the data without being excessively large, which could lead to overfitting. Last but not least, the output layer has 10 neurons, matching the 10 dimensions of the target output. In the development of our neural network, we have chosen the Adam optimizer and the Mean Squared Error (MSE) loss function for the compilation of our model. The rationale behind the choices of Adam, short for Adaptive Moment Estimation, is known for its effectiveness in handling sparse gradients and adapting the learning rate for each parameter. This feature is particularly beneficial for our dataset, where the range and scale of the data features might vary significantly.

6.7 Training

After simulating the robot's trajectory and assembling the dataset, our experiment's next phase focuses on the training process, which is critical for the model's performance. We have already discussed the dataset overview, data preprocessing, and the split into training and validation sets, along with the neural network architecture, in the previous subsection. Here, we delve deeper into the specifics of the training process, particularly concerning the batch size and learning rate, followed by an analysis of the training and testing results. Training is performed with a specified number of epochs and batch size, using a custom callback `TrainingMetricsCallback` for monitoring metrics during training. After training, the model is saved to a file for later use or deployment. In Figure.6.3, it presents the model performance across 60 epochs. Firstly, the model starts with a loss of 0.7141 and a validation loss of 0.6284, referring to Appendix.A.1. This relatively high loss indicates the model's initial inaccuracy in making predictions. After that, both the training loss and validation loss consistently decrease over epochs, which is a positive sign of learning. For instance, by epoch 10, the loss has reduced to 0.5978, and the val_loss to 0.6034, showing substantial improvement. By epoch 60 (Appendix Table.A.3), the model reaches its lowest loss (0.5696) and validation loss (0.5890). This gradual decrease suggest learning and generalization without significant overfitting. Moreover, the loss value hovering around 0.5698 in the final few epochs suggests that the model is approaching convergence. Convergence in this context means the model has reached a point where further training does not significantly reduce the loss, indicating it has found a stable set of weights that minimize the loss function. However, despite the fact that the loss values in the model stop decreasing and stabilize, it could mean the model has reached a plateau. This can be either a local or global minimum. The following context will confirm the model's generalization ability and evaluate it on a separate test dataset that wasn't used during training and validation. Furthermore, future experiments will also consider further hyperparameter tuning, like different types of optimizers or different architectures, which might lead to even better performances.

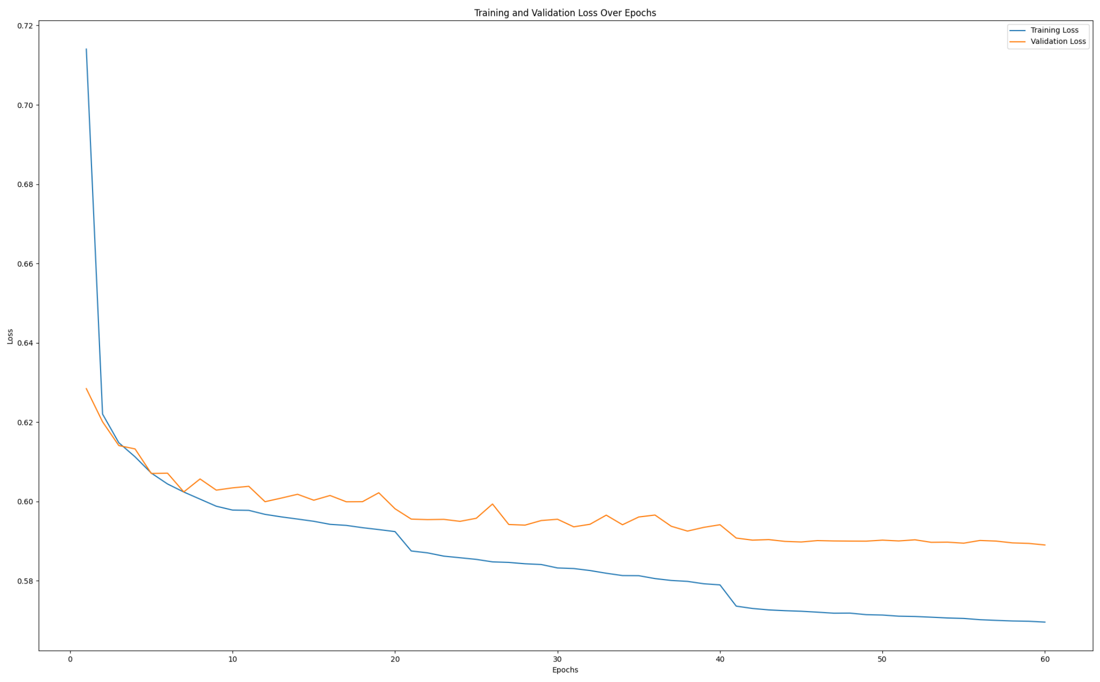

Figure 6.3: Factor graph representation of the learned smoothers

6.7.1 Batch size

Choice and justification: For our experiment, we experience different batch size ($e.g. 1, 4, 32, 64$), but ended up choosing a batch size of 32. The choice was motivated by the necessity to reduce calculation time during training. While a smaller batch size enables more frequent updates of model weights, resulting in a more finely tuned model, the training time for batch size 1 ranged from 130 to 220 seconds per epoch. In contrast, batch size 32 took only approximately 5 seconds per epoch, which is significantly shorter than batch size 1. Additionally, as shown in Table 6.1, the training with batch size 1 shows a similar final loss and validation loss compared to batch size 32. This suggests similar final loss values for both scenarios suggest that the model's performance is not significantly affected by the choice of batch size. Thus, the final selection for the training batch size is 32.

Table 6.1: Comparison of Training and Validation Losses for Batch Sizes 32 and 1

Epoch	Batch Size 32 Loss	Batch Size 1 Loss	Batch Size 32 Val Loss	Batch Size 1 Val Loss
1	0.7141	0.6523	0.6284	0.6279
2	0.6220	0.6194	0.6201	0.6208
3	0.6148	0.6151	0.6141	0.6114
...	...	...	...	...
59	0.5698	0.5859	0.5894	0.5949
60	0.5696	0.5859	0.5890	0.5947

6.7.2 Learning Rate

Dynamic adjustment: The learning rate in our model is not static; it is adjusted dynamically through a learning rate schedule. Initially set to 0.001, it decreases to 0.0005 after 20 epochs and further drops to 0.0001 beyond 40 epochs. This strategy helps in initially making updates to the model weights, which is beneficial for faster convergence, and then making smaller, more precise adjustments in later stages of training.

6.8 Testing

As demonstrated in the Appendix Coding Listing.A.4, the trained model is loaded from the file **trained_model.keras** using TensorFlow's `load_model` function. Predictions are then made on the normalized test set features using the model's `predict` method. The performance of the model is evaluated by calculating the Mean Squared Error (MSE, eq.6.12) and Mean Absolute Error (MAE eq.6.13) between the predicted labels $y_{\text{test\_pred}}$ and the true labels.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{6.12}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i| \tag{6.13}$$

where Y_i is the actual value, $\hat{Y}_i$ is the predicted value, and n is the number of testing samples. These metrics are used to assess the performance of models, with lower values indicating better model performance. The model is specifically built for the purpose of predicting the movements of robots. This task requires comprehending the connections between control inputs and subsequent changes in robot pose. The model must be able to capture both linear and non-linear processes. According to table.6.2, the findings from the non-Gaussian noise testset, exhibit a slightly better level of accuracy. This indicates that the model, which was trained on Gaussian noise, might be generalizing better to unseen data (in this case, Non-Gaussian Noise), which is an ideal situation. It implies that while the model was trained on Gaussian noise, it has learned patterns that are also applicable to other types of noise, potentially because of underlying similarities between the two noise types. Additionally, another factor to take into account for this outcome is that the model may be too simple to comprehend the intricacies of the data structure it was trained on.

Test Set	**Mean Squared Error (MSE)**	**Mean Absolute Error (MAE)**
Non-Gaussian Noise	0.6022	0.3968
Gaussian Noise (Baseline)	0.6136	0.4370

Table 6.2: Comparison of Model Performance on Non-Gaussian and Gaussian Noise Test Sets

6.9 Building Factor Graph by GTSAM Python wrapper

6.9.1 Variables & Key

Prior of delving into the SLAM system, we will present certain characteristics by utilising elements in GTSAM with Python. Factors represent the constraints or measurements relating to different variables (e.g., robot poses, landmark positions) in the graph. The variables within a factor graph correspond to the quantities that are being sought for estimation. In GTSAM, symbolic keys are commonly used to build these objects (e.g., for poses or landmarks), which are subsequently put into a Values container. The key does not need to be sequentially numbered, but it serves as the unique identification for either the robot's pose or a specific landmark in a given factor graph.

This implies that we have the capability to create the key for the robot's position and landmark in an arbitrary way. Here is an illustration (Code listing.6.4) demonstrating the process of creating variables using the key. In our project, we assign the key of the robot pose starts from 1 and the key of the landmark begins with 1000. In line 6 of this code snippet, it is initializing the state of the first pose, $(0, 0, 0)$ to be at the origin of the coordinate system with no rotation. 'Pose2' is a class in GTSAM that represents a 2D pose. A pose in this context is a combination of a position and orientation in a two-dimensional space. By passing $(0, 0, 0)$, this line is specifying that the initial estimate for the first pose has its position at the origin $(0, 0)$ of the coordinate system, and its orientation angle ψ is 0 radians, meaning it's aligned with the x-axis without any rotation.

```
1 # Define key ranges for poses and landmarks with distinct integer ranges
2 pose_key_counter = 1
3 landmark_key_counter = 1000  # Start landmark keys at a different range
4
5 first_pose_key = pose_key_counter
6 initial_estimate.insert(first_pose_key, gtsam.Pose2(0, 0, 0))
```

Listing 6.4: Create Variables

6.9.2 Noise model

Noise models in GTSAM are essential elements that describe the level of uncertainty related to measurements and constraints in a factor graph. These noise models are represented by the covariance matrix, as we previously described. They have a crucial function in the optimisation process by determining the appropriate level of 'confidence' to assign to each measurement or constraint in relation to others. This impacts the solution of the optimisation problem, which seeks to identify the optimal estimates for variables (such as robot positions and landmark positions) that most accurately align with the provided measurements, taking into account their associated uncertainties. In the code listing.6.5, a diagonal Gaussian noise of type **noiseModel.Diagonal** is provided for odometry by defining three standard deviations: 10 cm for the robot's location and 0.01 radians for the robot's orientation. The measurement noise model specifies two standard deviations: one of 10 cm for the range between the landmark and the robot's position, and one of 0.01 for the bearing angles between the landmark and the robot's posture.

```
#Define prior, odometry, measurement noise model
prior_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0.1, 0.1, 0.01]))
odometry_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0.1, 0.1,
    0.01]))
measurement_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0.1, 0.01]))
```

Listing 6.5: Noise Model

6.9.3 Factors

The factor graph representing a basic example is seen in Figure 5.1. The poses of the robot over time are represented by many variables, namely $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$, and so on. This figure shows a unary factor $f_0(\mathbf{x}_0)$ representing prior knowledge about the first posture $\mathbf{x}_0$, multiple binary factors connecting consecutive poses, and unary factors for the landmark measurement. GTSAM provides diverse factor types to accomplish the scenario depicted in Figure.5.1, including prior factors, between factors, and measurement factors. The code snippet (Listing.6.6) exemplified the utilisation of these factors.

```
# Create and add the prior factor for the first pose
prior_factor = gtsam.PriorFactorPose2(first_pose_key, initial_pose_guess,
    prior_noise)

# Create the motion factor (BetweenFactor)
motion_factor = gtsam.BetweenFactorPose2(current_key, next_key,
    nn_prediction_motion, noise_model)

# Add the bearing-range factor between new_pose_key and landmark_key
graph.add(gtsam.BearingRangeFactor2D(new_pose_key, landmark_key,
    measured_bearing, range, measurement_noise))
```

Listing 6.6: Factors

As shown in Listing.6.7, a **NonlinearFactorGraph** object is used to contain all the factors. The reason why GTSAM needs to perform non-linear optimization is because these factors are non-linear, as they involve the orientation of the robot.

```
graph = gtsam.NonlinearFactorGraph()
graph.add(prior_factor)
graph.add(motion_factor)
graph.add(observation_factor)
```

Listing 6.7: Build the Factor Graph

6.10 SLAM System

6.10.1 Optimization techniques in SLAM

An essential component of this project is the implementation of a simultaneous localization and mapping (SLAM) system, which is employed once the prediction of the relative transformation, observation, and information matrix have been obtained. In this study, we select the Levenberg-Marquardt (LM) optimizer, which is implemented to solve nonlinear least squares problems once the factor graph is built. SLAM problems in GTSAM are formulated as nonlinear least squares problems. The goal is to find the variable values that minimize the sum of the squared errors introduced by the constraints between variables and landmarks. The advantage of using LM compared with Gauss-Newton optimiser had been discussed in the previous section.2.4. To sum up, the key to LM's approach is its adaptive damping mechanism, which adjusts the influence of the gradient descent component based on the current error. This helps in navigating the error landscape more effectively by controlling the step size. If the error reduction is significant, the algorithm behaves more like Gauss-Newton, taking larger steps. If the error reduction is poor, the damping increases, making the algorithm take smaller, more cautious steps like gradient descent. Thus, LM is also known as a combination of both the Gauss-Newton algorithm and the gradient. GTSAM includes both Levenberg-Marquardt (LM) and Gauss Newton optimizers, as well as the ISAM2 (Incremental Smoothing and Mapping 2) method for addressing the optimisation in SLAM system. This algorithm is especially suitable for handling large-scale and real-time applications. Although ISAM2 is capable of handling real-time performance and is specifically built for incremental updates, making it more ideal for simulating real work environments. Nevertheless, this study aims to thoroughly investigate the complete dataset that is accessible, and the optimisation process is conducted on this dataset. This approach enhances the accuracy of integrating robot motion prediction and observation into robot state estimation, and real-time performance is not a crucial factor at this particular stage.

6.10.2 Integrating deep Learning for enhanced pose estimation

A Pose Graph SLAM system aims to concurrently estimate the robot's trajectory and the environment's map by utilising a graph-based representation. In a two-dimensional setting featuring a robot and many landmarks, the process of SLAM commences by establishing an initial estimation of the robot's position. The robot's current position is determined by estimating its new pose using information from its previous position and the control inputs, such as data from wheel encoders. In our study, the prediction motion from the neural network will be used for the odometry factor that is added to the factor graph between consecutive poses along with the associated uncertainty. This action will generate a new node (also known as a vertex) within the factor graph. Upon perceiving a landmark, the robot produces measurements that establish the relationship between its own position and the position of the landmark. This process involves the addition of a new landmark to the map or the adjustment of the position of an existing landmark. As a result, new connections are formed in the graph that link the robot's pose to these landmarks. This project investigates the use of deep learning neural networks to estimate the motion of a robot and the observations made between the robot and landmarks. The experiment is designed to correct the robot's motion and observations. This step calculates the expected observations for the predicted pose, compares them with the actual observations, and adjusts the pose to reduce the error.

6.10.3 SLAM simulation

The Appendix code listings A.6 (robot.py) and A.7 (SLAMSystem.py) define a framework for simulating a robot's interaction within a virtual 2D environment, incorporating SLAM capabilities enhanced by neural network predictions. Firstly, in code listing, A.6 structured around three main classes: **SimulatedEnvironment**, **SimulateRobot**, and **SLAMNNIntegrator**, each serving distinct roles in the simulation process. Firstly, the 'SimulatedEnvironment' read the file (landmark.txt) from the section.6.3.1 with 500 unique landmarks and their positions as values, these landmark's position was the same as the landmarks simulated in several trajectories with various wheel speeds and turning angles for collecting training samples. It is important to notice that in this class, we establish the key, or identity, of the landmark as starting at 1001. This key is needed to differentiate it from the robot's pose key, which starts at 1. As we construct the feature of the map, the 'SimulateRobot' constructs an object for simulating a robot's movement and perception within the 'SimulatedEnvironment'. There are three core functions defined for the robot: 'dynamic_model' and the 'observation_model' were discussed in the previous section (Sec.6.3.2, 6.3.3), which simulate the robot's motion given control inputs, update its pose, and simulate the detection of nearby landmarks within a 5 unit radius. Moreover, 'sense_odometry' simulates the robot obtaining odometry data from the dynamic model and updating the pose in GTSAM.Pose2. In addition to the functions defined the robot's behaviour, there are two functions, 'pose_error' and 'calculate_rmse' are designed to evaluate the accuracy of estimated poses against the true poses of both the robot and landmark position. The RMSE (Root Mean Square Error, eq.6.14) for both position and orientation errors can be represented as follows, where n is the total number of poses:

$$\begin{aligned} \text{RMSE}_{\text{position}} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left((x_{i,\text{true}} - x_{i,\text{est}})^2 + (y_{i,\text{true}} - y_{i,\text{est}})^2\right)} \\ \text{RMSE}_{\text{orientation}} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\arctan 2\left(\sin(\theta_{i,\text{true}} - \theta_{i,\text{est}}), \cos(\theta_{i,\text{true}} - \theta_{i,\text{est}})\right)\right)^2} \end{aligned} \tag{6.14}$$

Last but not least, 'SLAMNNIntegrator' integrates a trained neural network model to predict robot motion, observation, and information matrix, enhancing the SLAM process. According to the eq.5.6, the function of 'predict_motion_and_noise' uses the neural network (trained_model.keras) to predict the robot's next motion, observation, and information matrix based on the control inputs and the current robot's pose. This estimation will update the SLAM graph by adding the factor (BetweenFactorPose2 & BearingRangeFactor2D) to the graph.

In addition to the Appendix code listing A.6, the code listing A.7 is structured around simulating a robotic system that navigates an environment populated with landmarks as the main script to implement, enhanced by neural network predictions. In the core function 'create_slam_with_real_time_plotting_corrected', the script initializes the essential components for a SLAM system: a factor graph for modeling the robot's path and observations, initial estimates for the robot's poses, and various noise models to simulate real-world uncertainties in measurements and movements. The setup includes configuring real-time plotting to visualize the robot's path, landmarks, and their estimated positions dynamically. In the simulation, the robot sequentially adopts various poses to imitate movement, starting with a prior factor to anchor its initial position in the factor graph. It simulates movement through dynamic models, using a SLAM neural network integrator to predict motions and incorporate uncertainties, thus enhancing realism by adjusting initial estimates with new data. Additionally, it detects landmarks, integrating noisy observations of bearing and range into the graph, which aids in estimating landmark positions and refining the SLAM system's environmental model. This study does not include the integration of observational prediction and the information matrix derived from the neural network. However, we are still utilising the predefined observation model as a factor for robots and landmarks. Furthermore, real-time plotting is crucial for observing the SLAM process, allowing for instant evaluation of its accuracy by dynamically showing the true and estimated positions of the robot and landmarks. After constructing the factor graph, the script employs the Levenberg-Marquardt algorithm to optimize the SLAM graph, fine-tuning the robot's path and landmarks' positions to reduce error. The script concludes by calculating the RMSE for the robot's path accuracy and updating the plot with final estimates, showcasing the SLAM system's effectiveness and the enhancement potential through neural network integrations for accurate environment mapping and navigation.

Chapter 7

Results & Discussion

Our study seeks to evaluate the effect of incorporating neural network predictions into SLAM systems on the precision of SLAM results. This chapter explores the findings derived from the experiment, which involves comparing traditional SLAM methods with those improved by neural network predictions. We want to comprehensively examine the potential of neural networks in enhancing state estimation in the SLAM framework, using careful analysis accompanied by graphs and quantitative data. Moreover, this chapter will also discuss the limitations encountered during the study, and these discussions are crucial for future improvements and research.

7.1 Advancements and Challenges in SLAM Systems

7.1.1 Impact of wheel speed range on training performance

The table labeled 7.1 displays the training and validation loss outcomes for batch sizes of 1 and 32, but with variations in the wheel speed range. The wheel speed was adjusted to range from -1 to 1 metre per second in both the x and y directions, while the angular velocity was limited to a range of -0.1 to 0.1 radians per second. In comparison to the table labelled 6.1, where the wheel speed in the linear velocity direction ranged from 0 to 2 metres per second and the angular velocity ranged from 0 to 0.2, the experimental results indicate that the range of wheel speeds, although spanning the same 2 units, significantly affects training performance due to the presence of negative values in one scenario. When the wheel speed was constrained between -1 and 1 metres per second in both the x and y directions, and the angular velocity was restricted to -0.1 to 0.1 radians per second, the model demonstrated lower training and validation losses for batch sizes of 1 and 32. This improvement was observed in comparison to when the wheel speed range was adjusted to 0 to 2 metres per second, with angular velocity set to 0 to 0.2 radians per second. This distinction suggests that the inclusion of negative speeds allows the robot to move in multiple directions, rather than just in a positive direction. This enables the robot to gather a wider range of patterns and complexities for the model to learn. By incorporating negative directions, the training dataset is enriched with a wider range of scenarios, hence improving the model's capacity to make generalisations based on the data. Contrary to what was initially expected, merely increasing the range without including negative values may not enhance performance. This implies that the key to achieving superior training outcomes resides not in the width of the range, but rather in its composition. Future research could aim to investigate the expansion of the wheel speed range by integrating negative values. This would allow for a wider range of training samples, which could enhance the training performance in learning dynamics and improve model accuracy.

Table 7.1: Training and Validation Losses in different range of control input in batch size 32 and 1

Epoch	Batch Size 32 Loss	Batch Size 1 Loss	Batch Size 32 Val Loss	Batch Size 1 Val Loss
1	0.6894	0.6543	0.6451	0.6588
2	0.6324	0.6188	0.6369	0.6110
3	0.6209	0.5875	0.6195	0.5776
...	...	...	...	...
59	0.4049	0.3644	0.4272	0.3855
60	0.4043	0.3644	0.4274	0.3849

7.1.2 Evaluating the Impact of Neural Network Integration on SLAM Accuracy

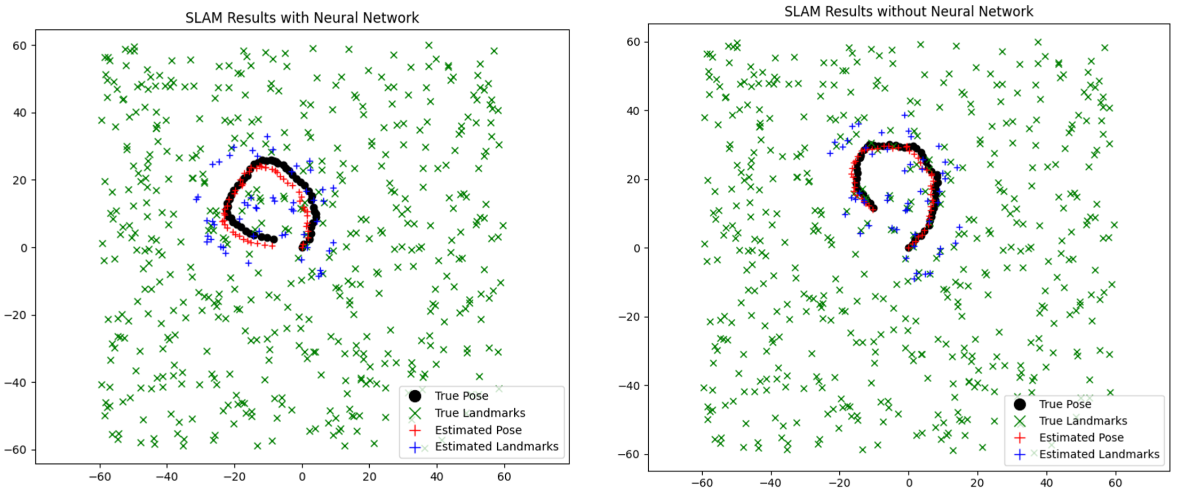

Figure 7.1: Comparative analysis of SLAM outcomes with and without the integration of a Neural Network

The provided figure.7.1 illustrates the outcomes of a SLAM experiment designed to evaluate the efficacy of integrating neural network predictions into the state estimation process. In this experiment, the initial robot pose was set at the origin $(0, 0, 0)$, with a constant wheel speed set to $(1, 1, 0.1)$. The graphical representation shows the true and estimated poses of the robot, as well as the positions of the landmarks.

Estimation	**Traditional SLAM RMSE**	**Neural Network SLAM RMSE**
Robot Pose	1.5219	2.9891
Landmark Position	0.0978	0.2154

Table 7.2: Comparison of RMSE values for traditional and neural network-augmented SLAM models

From the table of 7.2, the model without neural network predictions yielded an RMSE of 1.5219 for the robot's pose estimation and 0.0978 for landmark position estimation. In contrast, the model enhanced with neural network predictions resulted in a higher RMSE of 2.9891 for the robot's pose and 0.2154 for the landmarks. These results indicate a decrease in accuracy for the neural network-augmented model compared to the traditional SLAM approach. The traditional model appears to track the robot's trajectory with greater precision, as seen by the tighter clustering of estimated poses around the true path, depicted by the red plus signs closely following the black dots. Similarly, the estimated landmark positions (blue plus signs) are in closer proximity to the true landmark positions (green crosses) in the traditional model.

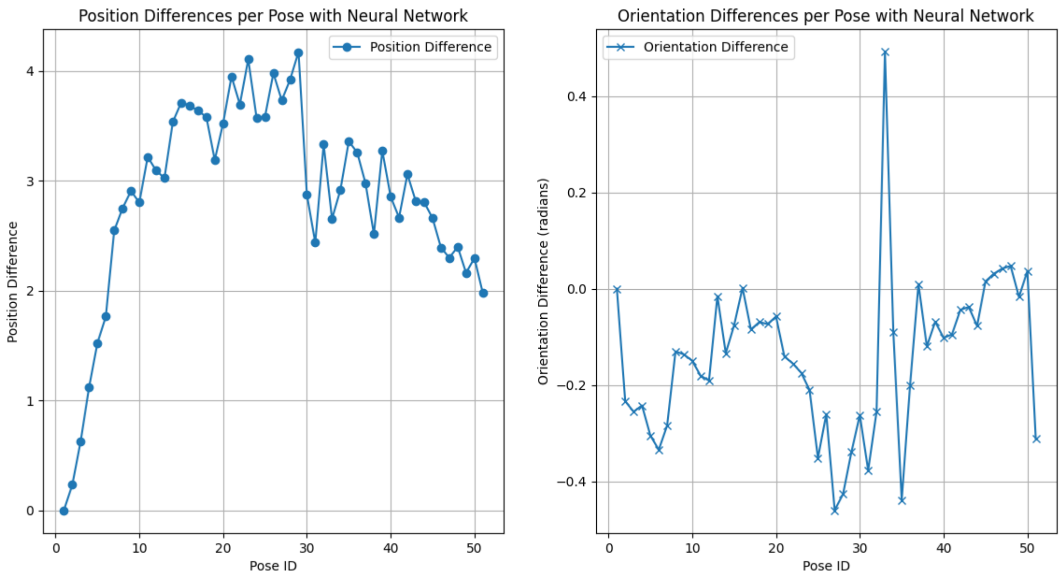

Figure 7.2: Position and orientation difference per pose with Neural network

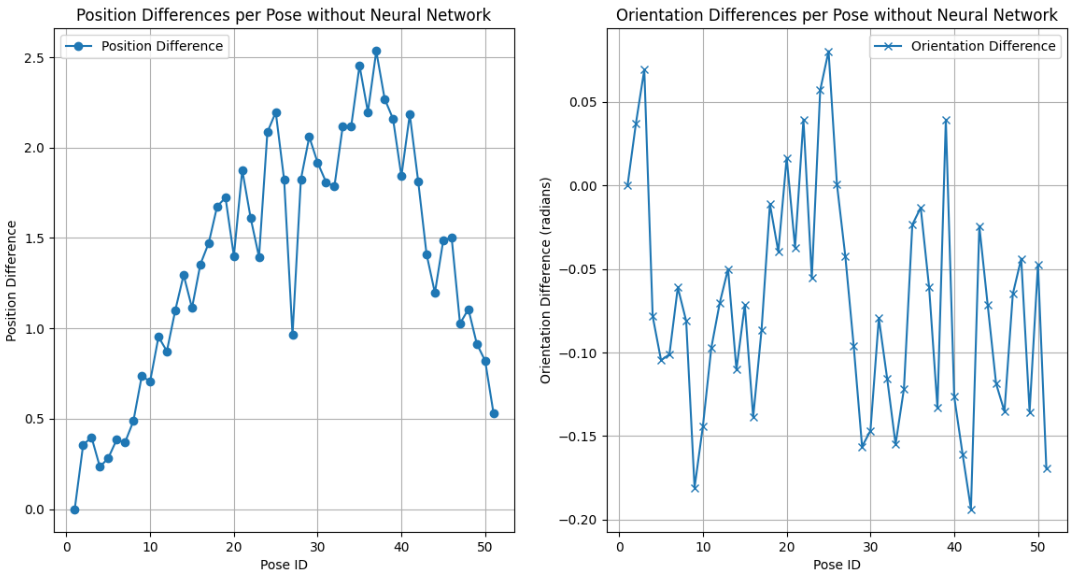

Figure 7.3: Position and orientation difference per pose without Neural network

The position difference graphs for both the neural network-enhanced model and the traditional model (Left Figure.7.2 & 7.3) display an increasing trend in position differences as the Pose ID increases. This trend potential suggests the presence of drift (Sec.3.1.1), a common issue in SLAM systems where error accumulates over time as the robot moves, leading to increasingly inaccurate pose estimates. The drift appears to be more pronounced in the neural network model, as evidenced by the higher position differences compared to the traditional model. This could indicate that the neural network's predictions are less stable or less accurate over time, potentially due to overfitting to the initial poses or an inability to generalize well to the entire trajectory. Additionally, the orientation difference graphs (Right Figure.7.2 & 7.3) show fluctuations in the orientation error for both models. However, the neural network model appears to have more extreme spikes in orientation differences, indicating moments of significant deviation from the true pose. These deviations could result from the neural network's predictions introducing additional noise or inaccuracies in the orientation estimates, which could be amplified over successive poses.

The comparison of X and Y coordinates (Figure.7.4 & 7.5) reveals how closely the estimated poses follow the true trajectory. The traditional model maintains a closer approximation to the true coordinates, with the estimated values closely tracking the black dots representing the true X and Y coordinates. In contrast, the neural network model exhibits a notable divergence from the true values.

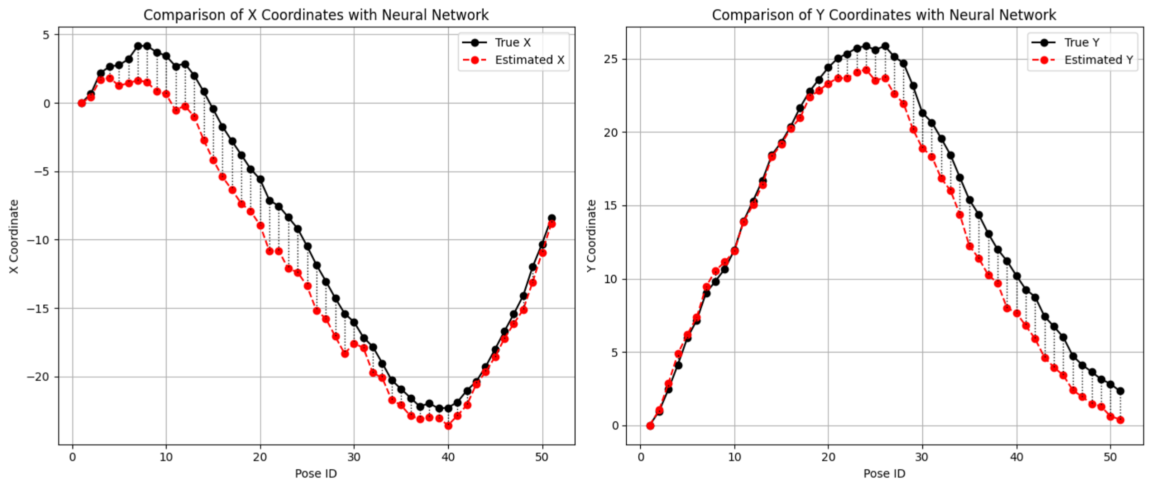

Figure 7.4: Comparison of X and Y coordinates with Neural network

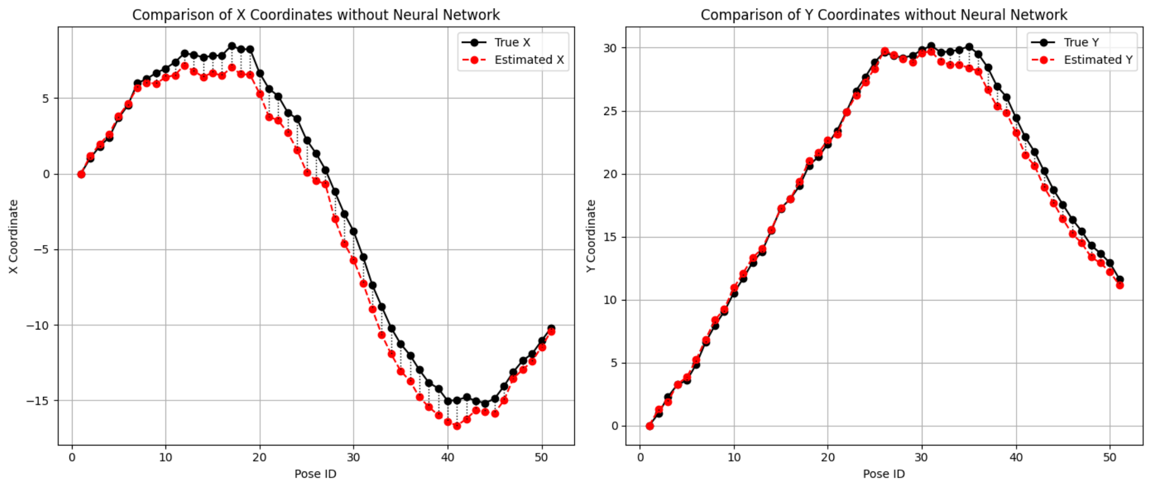

Figure 7.5: Comparison of X and Y coordinates without Neural network

The results suggest that the neural network-enhanced SLAM model does not outperform the traditional model. The increasing trend in position differences and the larger orientation errors point to a less accurate state estimation process when the neural network is used. This could be due to several factors, such as inadequate training data, improper network architecture, or a lack of robustness in the neural network's predictive capabilities.

7.2 Limitations

Building upon the preceding result and disuccsion, the visualisation indicates that, in contrast to what was anticipated, the neural network did not improve the performance of the SLAM system. The increased distribution of predicted robot postures and landmark positions in the neural network model, as evidenced by the wider scattering of red and blue plus signs (Figure.7.1), suggests a lower level of precision in state estimation. This unexpected outcome prompts inquiries regarding the training and appropriateness of the neural network architecture for this particular application. This suggests that the neural network may not have received sufficient training, possibly because its architecture is not sufficiently complex to effectively adapt to the test environment. Moreover, the significant increase in orientation indicates that the orientation robot fails to learn effectively because of the imprecise loss function in the training model. The future should explore different methods for handling the manifold structure instead of using Euclidean loss. Further, in gathering training samples, we do not collect numerous observations at each detection by the robot position but instead select the approach by choosing the nearest observation within the 5-radius unit as the robot centre. Hence, in future enhancements, incorporating the neural network's observation estimation into the SLAM system could potentially provide additional data to restrict the robot's position and landmark coordinates. Similarly, in our study, we remove the inclusion of the prediction of the information matrix from the neural network. our omission may affect the training model performance result during the final optimisation phase. In addition to the underlying factors that contributes to the inferior performance of the SLAM system with a neural network compared to the traditional SLAM, future research could incorporate loop closure in the trajectory to assess the performance outcome. This would ideally lead to a reduction in drift when the robot revisits past locations. In addition, it would have been beneficial to incorporate more experiments by employing diverse control inputs and unfamiliar landmark positions to observe the system's response. Additionally, it is worth considering the utilisation of ISAM2 algorithms in real-time scenarios, with the potential use of the system in the realm of three-dimensional environments.

Chapter 8

Conclusion

This study investigates the integration of the deep learning methodology with the SLAM system to acquire knowledge about robot motion, observations between landmarks and the robot position, and the information matrix. This information was crucial in situations when the process and observation noise were uncertain and difficult to specify in the analytical-based model. This commonly occurs when the system relies on data-rich modalities such as pictures, sound, or tactile feedback. To set up the experiment, we begin by constructing a neural network that can accurately capture the robot's mobility, observations, and the information matrix derived from the random wheel speed and robot posture. As the predcition of these information was learned, adopt the graphical model known as factor graph to address SLAM problem by utilising LM algorithms for optimisation. Although the results demonstrate that the system enhanced by neural network prediction does not surpass the traditional system, we have identified the potential issues responsible for this and have proposed a method to enhance the neural network model's understanding of the data structure. Additionally, for the future improvement, the study can consider adding the loop closure constraint and taken into account scenarios where the landmark may be unknown in the experiment in order to test the robust of the SLAM system with the integration of deep learning technique.

Bibliography

[1] Bohg, J., Kloss, A., & Martius, G. (2021). How to train your differentiable filter. *Autonomous Robots.*

[2] Bailey, T., & Durrant-Whyte, H. (2006). Simultaneous localization and mapping (SLAM): part II. In IEEE Robotics & Automation Magazine, vol. 13, no. 3, pp. 108-117.

[3] Bergman, N. (1999). Recursive Bayesian estimation: Navigation and tracking applications (Doctoral dissertation, Linköping University).

[4] Czarnowski, J., Laidlow, T., Clark, R., & Davison, A. J. (2020). DeepFactors: Real-Time Probabilistic Dense Monocular SLAM. *IEEE Robotics and Automation Letters, 5*(2), 721-728.

[5] Dellaert, F., & Kaess, M. (2017). Factor Graphs for Robot Perception. Now Publishers Inc.

[6] Dellaert, F. (2012). Factor graphs and GTSAM: A hands-on introduction. *Georgia Institute of Technology, Tech. Rep*, 2, 4.

[7] Dellaert, F., & Kaess, M. (2006). Square Root SAM: Simultaneous localization and mapping via square root information smoothing. The International Journal of Robotics Research, 25(12), 1181-1203.

[8] Durrant-Whyte, H., & Bailey, T. (2006). Simultaneous localization and mapping: part I. IEEE robotics & automation magazine, 13(2), 99-110.

[9] Dissanayake, G., Huang, S., Wang, Z., & Ranasinghe, R. (2011). A Review of Recent Developments in Simultaneous Localization and Mapping. In *2011 6th International Conference on Industrial and Information Systems*, pages 477-482. doi:10.1109/ICIINFS.2011.6038117

[10] Eade, E. (2013). Gauss-newton/levenberg-marquardt optimization. Tech. Rep.

[11] György, K., Kelemen, A., & Dávid, L. (2014). Unscented Kalman Filters and Particle Filter Methods for Nonlinear State Estimation. *Procedia Technology, 12*, 65-74. ISSN 2212-0173.

[12] Grisetti, G., Kümmerle, R., Strasdat, H., & Konolige, K. (2011, May). g2o: A general framework for (hyper) graph optimization. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, (pp. 9-13).

[13] Gao, X., & Zhang, T. (2021). Introduction to visual SLAM: from theory to practice. *Springer Nature.*

[14] Gavin, H. P. (2019). The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems. Department of Civil and Environmental Engineering, Duke University, 19.

[15] Haarnoja, T., Ajay, A., Levine, S., & Abbeel, P. (2017). Backprop KF: Learning Discriminative Deterministic State Estimators. arXiv preprint arXiv:1605.07148.

[16] Hess, W., Kohler, D., Rapp, H., & Andor, D. (2016). Real-time loop closure in 2D LIDAR SLAM. In 2016 IEEE International Conference on Robotics and Automation (ICRA) (pp. 1271-1278).

[17] Holl, P., Koltun, V., & Thuerey, N. (2020). Learning to Control PDEs with Differentiable Physics. In *arXiv preprint arXiv:2001.07457.*

[18] Hussain, R.; Zeadally, S. (2018). Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys & Tutorials*, 21, 1275–1313.

[19] Huang, R.; Zhuo, G.; Xiong, L.; Lu, S.; Tian, W. (2023). A Review of Deep Learning-Based Vehicle Motion Prediction for Autonomous Driving. *Sustainability*, 15(20), 14716. `https://doi.org/10.3390/su152014716`

[20] Jonschkowski, R., & Brock, O. (2017). End-to-End Learnable Histogram Filters.

[21] Jatavallabhula, K. M., Saryazdi, S., Iyer, G., & Paull, L. (2020). gradSLAM: Automagically differentiable SLAM. arXiv preprint arXiv:1910.10672.

[22] Julier, S. (2023). Sparse Approaches to SLAM [Lecture slides]. COMP0130: Robot Vision and Navigation, University College London.

[23] Jonschkowski, R., Rastogi, D., & Brock, O. (2018). Differentiable Particle Filters: End-to-End Learning with Algorithmic Priors. arXiv preprint arXiv:1805.11122.

[24] Karkus, P., Hsu, D., & Lee, W. S. (2018). Particle Filter Networks with Application to Visual Localization. arXiv preprint arXiv:1805.08975.

[25] Karkus, P., Ma, X., Hsu, D., Kaelbling, L. P., Lee, W. S., & Lozano-Perez, T. (2019). Differentiable Algorithm Networks for Composable Robot Learning. In arXiv preprint arXiv:1905.11602.

[26] Liu, J., & West, M. (2001). Combined parameter and state estimation in simulation-based filtering. In: *Sequential Monte Carlo methods in practice* (pp. 197-223). Springer.

[27] Pernice, M., & Walker, H. F. (1998). NITSOL: A Newton iterative solver for nonlinear systems. *SIAM Journal on Scientific Computing, 19*(1), 302-318.

[28] Rabiee, S., & Biswas, J. (2020). IV-SLAM: Introspective Vision for Simultaneous Localization and Mapping. arXiv preprint arXiv:2008.02760.

[29] Sodhi, P., Kaess, M., Mukadam, M., & Anderson, S. (2021). Learning Tactile Models for Factor Graph-based Estimation. arXiv preprint arXiv:2012.03768.

[30] Schmidt, M., Berg, E., Friedlander, M., & Murphy, K. (2009, April). Optimizing costly functions with simple constraints: A limited-memory projected quasi-newton algorithm. In *Artificial intelligence and statistics* (pp. 456-463). PMLR.

[31] Strasdat, H., Montiel, J. M., & Davison, A. J. (2012). Visual SLAM: why filter?. Image and Vision Computing, 30(2), 65-77.

[32] Smith, R.C., & Cheeseman, P. (1986). On the Representation and Estimation of Spatial Uncertainty. *The International Journal of Robotics Research*, 5(4), 56-68. doi:10.1177/027836498600500404

[33] Tang, C., & Tan, P. (2019). BA-Net: Dense Bundle Adjustment Network. arXiv preprint arXiv:1806.04807.

[34] Thrun, S. (2002, August). Particle Filters in Robotics. In UAI (Vol. 2, pp. 511-518).

[35] Tamar, A., Wu, Y., Thomas, G., Levine, S., & Abbeel, P. (2017). Value Iteration Networks. In *arXiv preprint arXiv:1602.02867.*

[36] Welch, G., & Bishop, G. (1995). An introduction to the Kalman filter. Chapel Hill, NC, USA.

[37] Wu, M., Cheng, C., & Shang, H. (2021, November). 2D LIDAR SLAM Based On Gauss-Newton. In 2021 International Conference on Networking Systems of AI (INSAI) (pp. 90-94). IEEE.

[38] Yi, B., Lee, M., Kloss, A., Martín-Martín, R., & Bohg, J. (2021). Differentiable Factor Graph Optimization for Learning Smoothers. *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*

Appendix A

A.1 Code Listings

```
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_title('Robot Movement Over Time with Landmarks and Distances')
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    ax.grid(True)

    line, = ax.plot(trajectory[0][0], trajectory[0][1], label='Robot Trajectory', marker='o')
    scatter_landmarks = ax.scatter(landmarks[:, 0], landmarks[:, 1], color='red', marker='*', label='Landmarks')

    min_x, max_x = np.min(landmarks[:, 0]), np.max(landmarks[:, 0])
    min_y, max_y = np.min(landmarks[:, 1]), np.max(landmarks[:, 1])
    ax.set_xlim(min_x - 5, max_x + 5)
    ax.set_ylim(min_y - 5, max_y + 5)
    fig.canvas.draw()

    # Enable interactive zoom
    fig.canvas.mpl_connect('scroll_event', lambda event: on_scroll(event, ax))
    # Uncomment the following lines to display connections between robot and landmarks
    # con = ConnectionPatch(xyA=(x_k[0], x_k[1]), xyB=(landmark[0], landmark[1]),
    #                       coordsA="data", coordsB="data", color="blue", arrowstyle="->", linewidth=2)
    # ax.add_patch(con)
```

Listing A.1: Matplotlib code for plotting robot movement, landmarks, and distances.

```
def save_to_csv(filename, acc_step, trajectory, control_inputs, relative_poses_list, obs_list, covariances, write_header):
    mode = 'w' if write_header else 'a'  # Open in write mode if writing header, else append mode
    with open(filename, mode, newline='') as csvfile:
        writer = csv.writer(csvfile)
        # Write header
        if write_header :
            writer.writerow(['Step', 'pose_x', 'pose_y', 'pose_theta', '
                u_k_x', 'u_k_y', 'u_k_theta', 'relative_X', 'relative_Y', '
                relative_Theta', 'Obs_dist', 'Obs_tetha', 'Covariance_X', '
                Covariance_Y', 'Covariance_Theta','Covariance_dis', '
                Covariance_angle'])
        # Write data
        for step, (st, relative_pose, obs, covariance, control_inputs, traj
            ) in enumerate(zip(acc_step, relative_poses_list, obs_list,
            covariances, control_inputs, trajectory)):

            # Format covariance values with 4 decimal places
            formatted_covariance_x = f"{covariance[0][0]:.4f}"
            formatted_covariance_y = f"{covariance[0][1]:.4f}"
            formatted_covariance_theta = f"{covariance[0][2]:.4f}"
            formatted_covariance_d = f"{covariance[1][0]:.4f}"
            formatted_covariance_t = f"{covariance[1][1]:.4f}"

            obs_d = f"{obs[0]:.4f}"
            obs_t = f"{obs[1]:.4f}"

            control_inputs_x = f"{control_inputs[0]:.4f}"
            control_inputs_y = f"{control_inputs[1]:.4f}"
            control_inputs_theta = f"{control_inputs[2]:.4f}"

            relative_pose_x = f"{relative_pose[0]:.4f}"
            relative_pose_y = f"{relative_pose[1]:.4f}"
            relative_pose_t = f"{relative_pose[2]:.4f}"

            trajectory_x = f"{traj[0]:.4f}"
            trajectory_y = f"{traj[1]:.4f}"
            trajectory_theta = f"{traj[2]:.4f}"

            writer.writerow([st, trajectory_x, trajectory_y,
                trajectory_theta, control_inputs_x, control_inputs_y,
                control_inputs_theta , relative_pose_x, relative_pose_y,
                relative_pose_t,  obs_d , obs_t , formatted_covariance_x,
                formatted_covariance_y, formatted_covariance_theta,
                formatted_covariance_d, formatted_covariance_t])
```

Listing A.2: Python function for saving data to a CSV file, including header management and data formatting.

```
# Load dataset from CSV file
# Replace 'dataset.csv' with the actual path to your CSV file
df = pd.read_csv('Dataset/dataset.csv')

# CSV has columns: 'Step', 'pose_x', 'pose_y', 'pose_theta', 'u_k_x', '
    u_k_y', 'u_k_theta', 'relative_X', 'relative_Y', 'relative_Theta', '
    Obs_dist', 'Obs_tetha', 'Covariance_X', 'Covariance_Y', '
    Covariance_Theta','Covariance_dis', 'Covariance_angle'
X = df[['pose_x', 'pose_y', 'pose_theta', 'u_k_x', 'u_k_y', 'u_k_theta']].
    values
y = df[['relative_X', 'relative_Y', 'relative_Theta',
        'Obs_dist', 'Obs_tetha',
        'Covariance_X', 'Covariance_Y', 'Covariance_Theta',
        'Covariance_dis', 'Covariance_angle']].values

# Normalize the data
scaler_X = StandardScaler().fit(X)
X_scaled = scaler_X.transform(X)

# Split into training and validation sets (80% train, 20% validation)
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y, test_size
    =0.2, random_state=42)

# Define the neural network model
model = tf.keras.Sequential([
    layers.Input(shape=(6,)),  # Input layer has 6 dimensions
    layers.Dense(64, activation='relu'), # Consider regularization
        techniques: kernel_regularizer=regularizers.l2(0.01)
    layers.Dense(64, activation='relu'),
    layers.Dense(32, activation='relu'),
    layers.Dense(10)  # Output has 10 dimensions
])

# Compile the model
learning_rate = 0.001 # Experiment with different values (e.g. 0.1, 0.01,
    0.001, 0.0005)
optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
model.compile(optimizer='adam', loss='mean_squared_error')

# Define the learning rate schedule
def learning_rate_schedule(epoch, lr):
    if epoch < 30:
        return 0.001
    else:
        return 0.0005

class TrainingMetricsCallback(tf.keras.callbacks.Callback):
    def __init__(self):
        super().__init__()
        self.train_loss = []
        self.val_loss = []

    def on_epoch_end(self, epoch, logs=None):
        # Store the loss values
        self.train_loss.append(logs.get('loss'))
        self.val_loss.append(logs.get('val_loss'))
```

```python

        # Print the metrics
        print(f"\nEpoch {epoch + 1}/{self.params['epochs']}")
        for metric_name, value in logs.items():
            print(f"{metric_name}: {value:.4f}")

    def on_train_end(self, logs=None):
        # Plot loss graph
        plt.figure(figsize=(10, 5))
        plt.plot(range(1, len(self.train_loss) + 1), self.train_loss, label='Training Loss')
        plt.plot(range(1, len(self.val_loss) + 1), self.val_loss, label='Validation Loss')
        plt.title('Training and Validation Loss Over Epochs')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()

# Adjust the learning rate during training
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(learning_rate_schedule)

# Plot the loss graph
metrics_callback = TrainingMetricsCallback()

# Train the model and tune epochs and batch size
model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50, batch_size=1, callbacks=[metrics_callback, lr_scheduler])

model.save('trained_model.keras')
```

Listing A.3: Python code for loading a dataset, preprocessing, defining, and training a neural network model by Tensorflow.

```python
# Load the test dataset
test_df = pd.read_csv('testset.csv')

# Extract features and labels for the test set
X_test = test_df[['u_k_x', 'u_k_y', 'u_k_theta', 'Obs_dist', 'Obs_tetha']].values
y_test_true = test_df[['relative_X', 'relative_Y', 'relative_Theta', 'Covariance_X', 'Covariance_Y', 'Covariance_Theta','Covariance_dis', 'Covariance_angle']].values

# Normalize the data
scaler_X = StandardScaler().fit(X_test)
X_scaled = scaler_X.transform(X_test)

# Load the trained model
```

```
model = tf.keras.models.load_model('trained_model.keras')

# Predictions on the test set
y_test_pred = model.predict(X_scaled)

# Calculate Mean Squared Error and Mean Absolute Error
mse = mean_squared_error(y_test_true, y_test_pred)
mae = mean_absolute_error(y_test_true, y_test_pred)

# Display Metrics
print("Mean Squared Error (MSE):", mse)
print("Mean Absolute Error (MAE):", mae)
```

Listing A.4: Test script

```
def normalize_theta(theta):
    if theta < -np.pi:
        theta += 2 * np.pi
    elif theta > np.pi:
        theta -= 2 * np.pi
    return theta
```

Listing A.5: Function to Normalize the Angle

```
class SimulatedEnvironment:
    def __init__(self, num_landmarks = 500, area_size=(200, 200)):
        self.landmarks = self.generate_landmarks(num_landmarks, area_size)

    def generate_landmarks(self, num_landmarks, area_size):
        # Generate landmarks within the specified area
        landmarks = {}
        filename = "landmarks.txt"
        with open(filename, 'r') as file:
            for i, line in enumerate(file):
                x, y = map(float, line.strip().split(','))
                landmarks[i+1000] = np.array([x, y])
        return landmarks

class SimulatedRobot:
    def __init__(self, environment):
        self.last_x = 0
        self.last_y = 0
        self.last_theta = 0
        self.environment = environment
        self.pose = gtsam.Pose2(0, 0, 0)  # Initial pose
        self.speed = 0.5  # Constant speed
        self.angular_velocity = 0.1  # Constant angular velocity
        self.delta_t = 1
        self.R_std_dev = np.diag([0.2, 0.05])

    def rotation_matrix(self, psi):
        return np.array([
            [np.cos(psi), -np.sin(psi), 0],
            [np.sin(psi), np.cos(psi), 0],
            [0, 0, 1]
        ])

    def normalize_theta(self, theta):
        if theta < -np.pi:
            theta += 2 * np.pi
        elif theta > np.pi:
            theta -= 2 * np.pi
        return theta

    def dynamic_model(self, x_k1, x_k2, psi_k, u_k):
        max_delta_x = 5  # max change in x
        max_delta_y = 5  # max change in y
        max_delta_theta = np.radians(30)
        Q_std_dev = np.diag([0.1, 0.1, 0.01])
        v_k = np.random.multivariate_normal([0, 0, 0], Q_std_dev)

        M_psi_k = self.rotation_matrix(psi_k)
        a = M_psi_k @ (u_k.T + v_k.T)
        delta = self.delta_t * a  # Calculate the change

        # Apply limits to delta
        delta[0] = np.clip(delta[0], -max_delta_x, max_delta_x)
        delta[1] = np.clip(delta[1], -max_delta_y, max_delta_y)
        delta[2] = np.clip(delta[2], -max_delta_theta, max_delta_theta)

        # Calculate new pose components
        new_x = x_k1 + delta[0]
        new_y = x_k2 + delta[1]
        new_psi = psi_k + delta[2]
        new_psi = self.normalize_theta(new_psi)  # Ensure psi remains
            within [-pi, pi]

        delta_x = new_x - self.last_x
        delta_y = new_y - self.last_y
        delta_theta = new_psi - self.last_theta
        delta_theta = self.normalize_theta(delta_theta)  # Ensure
            delta_theta is normalized

        # Apply movement limits
        delta_x = np.clip(delta_x, -max_delta_x, max_delta_x)
        delta_y = np.clip(delta_y, -max_delta_y, max_delta_y)
        delta_theta = np.clip(delta_theta, -max_delta_theta,
            max_delta_theta)

        # Calculate corrected new position based on limited deltas
        corrected_new_x = self.last_x + delta_x
```

```python
        corrected_new_y = self.last_y + delta_y
        corrected_new_psi = self.normalize_theta(self.last_theta + delta_theta)

        # Update the robot's pose to the corrected new position
        self.pose = gtsam.Pose2(corrected_new_x, corrected_new_y, corrected_new_psi)

        # Update last position and orientation
        self.last_x = corrected_new_x
        self.last_y = corrected_new_y
        self.last_theta = corrected_new_psi

        return gtsam.Pose2(delta_x,delta_y,delta_theta)

    def sense_odometry(self):
        # Simulate odometry data with noise
        noise_x, noise_y, noise_theta = np.random.normal(0, 0.1), np.random.normal(0, 0.1), np.random.normal(0, 0.05)
        noisy_pose = gtsam.Pose2(self.pose.x() + noise_x , self.pose.y() + noise_y  , self.pose.theta() + noise_theta )
        return np.array([noisy_pose.x() , noisy_pose.y(), noisy_pose.theta()])

    def observation_model(self, x_i, y_i, x_k, y_k, phi_k):
        # Simulate landmark observation with noise
        w_k = np.random.multivariate_normal([0, 0], self.R_std_dev)
        # Calculate range and bearing
        r_k_i = np.sqrt((x_i - x_k)**2 + (y_i - y_k)**2)  + w_k[0]
        beta_k_i = np.arctan2((y_i - y_k), (x_i - x_k)) - phi_k + w_k[1]
        return np.array([r_k_i, beta_k_i])

    def estimate_landmark_position(self, pose, bearing, landmark_range):
        # Implement landmark position estimation based on the pose, bearing, and landmark range
        x = pose.x() + landmark_range * np.cos(pose.theta() + bearing)
        y = pose.y() + landmark_range * np.sin(pose.theta() + bearing)
        return gtsam.Point2(x, y)

    def pose_error(self,true_pose, estimated_pose):
        """
        Calculate squared position and orientation errors between two poses.
        """

        position_error_sq = (true_pose[0] - estimated_pose[0])**2 + (true_pose[1]- estimated_pose[1])**2
        orientation_error = np.abs(true_pose[2] - estimated_pose[2])
        orientation_error = np.arctan2(np.sin(orientation_error), np.cos(orientation_error))  # Normalize to [-pi, pi]
        orientation_error_sq = orientation_error**2
        return position_error_sq, orientation_error_sq

    def calculate_rmse(self, true_poses, estimated_poses):
        """
        Calculate the RMSE for position and orientation across all poses.
        """
        position_error_sums = 0
        orientation_error_sums = 0
        n = len(true_poses)

        for id, true_pose in true_poses.items():
            if id in estimated_poses:
                estimated_pose = estimated_poses[id]
                pos_err_sq, ori_err_sq = self.pose_error(true_pose,
                    estimated_pose)
                position_error_sums += pos_err_sq
                orientation_error_sums += ori_err_sq
            else:
                print(f"Missing estimated pose for ID: {id}")
                n -= 1  # Adjust count if any poses are missing

        position_rmse = np.sqrt(position_error_sums / n)
        orientation_rmse = np.sqrt(orientation_error_sums / n)
        return position_rmse, orientation_rmse

class SLAMNNIntegrator:
    def __init__(self, model_path):

        self.model = tf.keras.models.load_model(model_path)

    def predict_motion_and_noise(self, u_k, x, y, theta):
        """
        Predicts the motion and noise covariance based on robot control
            inputs and a single observation.
        u_k: Control inputs array containing [velocity_x, velocity_y,
            angular_velocity].
        x, y, theta: Current robot pose.
        """
        # Assuming the inputs are already scaled/normalized if necessary
        combined_inputs = np.concatenate((u_k, np.array([x, y, theta])))
        combined_inputs = combined_inputs.reshape(1, -1)  # Reshape for the
             neural network

        predictions = self.model.predict(combined_inputs)[0]
        motion = gtsam.Pose2(predictions[:3] ) # Assuming the first three
            predictions are motion (x, y, theta)
        land_b_d =  predictions[3:5]
        noise_cov = predictions[5:]  # Assuming the rest are noise
            covariance
        return motion, land_b_d, noise_cov
```

Listing A.6: Neural Network Enhanced SLAM Simulation

```
# Function to calculate the smallest difference between two angles
def angle_difference(angle1, angle2):
    diff = (angle1 - angle2 + np.pi) % (2 * np.pi) - np.pi
    return diff

# Simulation parameters
num_landmarks_to_generate = 500

# Initialize simulated environment and robot
environment = SimulatedEnvironment(num_landmarks=num_landmarks_to_generate)
robot = SimulatedRobot(environment)

# Initialize SLAM Neural Network Integrator with a pre-trained model
model_path = 'trained_model.keras'
slam_nn_integrator = SLAMNNIntegrator(model_path)

# Main function for executing SLAM with real-time plotting
def create_slam_with_real_time_plotting_corrected(num_poses=50):
    # Initialize SLAM components: factor graph, initial estimates, and
        noise models
    graph = gtsam.NonlinearFactorGraph()
    initial_estimate = gtsam.Values()
    prior_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0.2, 0.2,
        0.1]))
    odometry_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0.3, 0.3,
        0.1]))
    measurement_noise = gtsam.noiseModel.Diagonal.Sigmas(np.array([0.1,
        0.1]))

    # Setup for real-time plotting
    plt.figure(figsize=(6, 6))
    plt.ion()

    # Initialize true and estimated poses dictionaries
    landmarks = environment.landmarks
    true_poses = {}
    estimated_poses = {}

    use_nn = input("Use neural network? (yes/no): ").strip().lower() == '
        yes'

    # Add a prior on the first pose
    first_pose = gtsam.Pose2(0.0, 0.0, 0.0)
    graph.add(gtsam.PriorFactorPose2(1, first_pose, prior_noise))
    initial_estimate.insert(1, first_pose)

    # Loop through the poses
    for i in range(1, num_poses + 2):
        if i == 1:
            new_pose = first_pose
        else:
            # Predict next pose based on previous pose and control inputs
            prev_pose = initial_estimate.atPose2(i-1)
            u_k = np.array([1, 1, 0.1])  # Example control inputs
            odometry = robot.dynamic_model(prev_pose.x(), prev_pose.y(), prev_pose.theta(), u_k)
            if use_nn:
                odometry, land, noise_cov = slam_nn_integrator.predict_motion_and_noise(u_k, prev_pose.x(), prev_pose.y(), prev_pose.theta())
            x_k2 = robot.sense_odometry()
            new_pose = gtsam.Pose2(x_k2[0], x_k2[1], x_k2[2])
            graph.add(gtsam.BetweenFactorPose2(i-1, i, odometry, odometry_noise))
            initial_estimate.insert(i, new_pose)

        # Simulate landmark observation and update graph
        for lm_id, lm_pos in landmarks.items():
            obs = robot.observation_model(lm_pos[0], lm_pos[1], new_pose.x(), new_pose.y(), new_pose.theta())
            if obs[0] < 10:
                graph.add(gtsam.BearingRangeFactor2D(i, lm_id, gtsam.Rot2(obs[1]), obs[0], measurement_noise))
                lnd_po = robot.estimate_landmark_position(new_pose, obs[1], obs[0])
                if initial_estimate.exists(lm_id):
                    updates = gtsam.Values()
                    updates.insert(lm_id, lnd_po)
                    initial_estimate.update(updates)
                else:
                    initial_estimate.insert(lm_id, lnd_po)

        # Update dictionaries with true and estimated poses
        true_poses[i] = [new_pose.x(), new_pose.y(), new_pose.theta()]

        # Real-time plotting of true poses and landmarks
        plt.clf()
        for j in range(1, i + 1):
            if j in initial_estimate.keys():
                pose = initial_estimate.atPose2(j)
                plt.plot(pose.x(), pose.y(), 'ko')  # Black circles for true poses
        for lm_id, lm_pos in landmarks.items():
            plt.plot(lm_pos[0], lm_pos[1], 'gx')

        plt.axis('equal')
        plt.draw()
        plt.pause(.001)

    # Optimize the factor graph after all observations
    optimizer = gtsam.LevenbergMarquardtOptimizer(graph, initial_estimate)
    result = optimizer.optimize()

    # Final update and plot of estimated poses and landmarks
    for i in range(1, num_poses + 2):
        if i in result.keys():
            estimated_pose = result.atPose2(i)
            estimated_poses[i] = [estimated_pose.x(), estimated_pose.y(), estimated_pose.theta()]
            plt.plot(estimated_pose.x(), estimated_pose.y(), ’r+’)  # Plot estimated pose as red pluses

    for lm_id in landmarks.keys():
        if lm_id in result.keys():
            estimated_landmark = result.atPoint2(lm_id)
            plt.plot(estimated_landmark[0], estimated_landmark[1], ’b+’)  # Plot estimated landmarks as blue pluses

    # Add legend to the plot
    true_pose_handle = mlines.Line2D([], [], color=’black’, marker=’o’, linestyle=’None’, markersize=10, label=’True Pose’)
    true_landmark_handle = mlines.Line2D([], [], color=’green’, marker=’x’, linestyle=’None’, markersize=10, label=’True Landmarks’)
    estimated_pose_handle = mlines.Line2D([], [], color=’red’, marker=’+’, linestyle=’None’, markersize=10, label=’Estimated Pose’)
    estimated_landmark_handle = mlines.Line2D([], [], color=’blue’, marker=’+’, linestyle=’None’, markersize=10, label=’Estimated Landmarks’)
    plt.legend(handles=[true_pose_handle, true_landmark_handle, estimated_pose_handle, estimated_landmark_handle])

    # Calculate and print RMSE for position and orientation
    position_rmse, orientation_rmse = robot.calculate_rmse(true_poses, estimated_poses)
    print("errors", position_rmse, orientation_rmse)

    # Plot differences in position and orientation for each pose
    plt.ioff()  # Turn off interactive mode
    plt.show()
    plt.pause(2)
    plt.clf()

    # Additional plots for position and orientation differences
    pose_ids = true_poses.keys()  # Assuming both dictionaries have the same keys
    position_differences = [np.sqrt((true_poses[id][0] - estimated_poses[id][0])**2 + (true_poses[id][1] - estimated_poses[id][1])**2) for id in pose_ids]
    orientation_differences = [angle_difference(true_poses[id][2],
        estimated_poses[id][2]) for id in pose_ids]

    # Position difference plot
    plt.figure(figsize=(14, 7))
    plt.subplot(1, 2, 1)
    plt.plot(list(pose_ids), position_differences, label='Position
        Difference', marker='o')
    plt.xlabel('Pose ID')
    plt.ylabel('Position Difference')
    plt.title('Position Differences per Pose')
    plt.grid(True)
    plt.legend()

    # Orientation difference plot
    plt.subplot(1, 2, 2)
    plt.plot(list(pose_ids), orientation_differences, label='Orientation
        Difference', marker='x')
    plt.xlabel('Pose ID')
    plt.ylabel('Orientation Difference (radians)')
    plt.title('Orientation Differences per Pose')
    plt.grid(True)
    plt.legend()
    plt.show()

    # Comparison plots for X and Y coordinates
    pose_ids = list(true_poses.keys())  # List of pose IDs for indexing
    true_x = [true_poses[id][0] for id in pose_ids]
    true_y = [true_poses[id][1] for id in pose_ids]
    estimated_x = [estimated_poses[id][0] for id in pose_ids]
    estimated_y = [estimated_poses[id][1] for id in pose_ids]

    # Plotting
    plt.figure(figsize=(14, 6))

    # Subplot for x coordinates
    plt.subplot(1, 2, 1)
    plt.plot(pose_ids, true_x, 'ko-', label='True X')
    plt.plot(pose_ids, estimated_x, 'ro--', label='Estimated X')
    for i, pid in enumerate(pose_ids):
        plt.plot([pid, pid], [true_x[i], estimated_x[i]], 'k:', linewidth
            =1)  # Dotted line between true and estimated

    plt.xlabel('Pose ID')
    plt.ylabel('X Coordinate')
    plt.title('Comparison of X Coordinates')
    plt.legend()
    plt.grid(True)

    # Subplot for y coordinates
    plt.subplot(1, 2, 2)
    plt.plot(pose_ids, true_y, 'ko-', label='True Y')
    plt.plot(pose_ids, estimated_y, 'ro--', label='Estimated Y')
    for i, pid in enumerate(pose_ids):
        plt.plot([pid, pid], [true_y[i], estimated_y[i]], 'k:', linewidth
            =1)  # Dotted line between true and estimated

    plt.xlabel('Pose ID')
    plt.ylabel('Y Coordinate')
    plt.title('Comparison of Y Coordinates')
    plt.legend()
    plt.grid(True)

    plt.tight_layout()
    plt.show()

create_slam_with_real_time_plotting_corrected()
```

Listing A.7: SLAM system

A.2 Additional Table

Epoch	Loss	Validation Loss	Learning Rate
1	0.7141	0.6284	0.0010
2	0.6220	0.6201	0.0010
3	0.6148	0.6141	0.0010
4	0.6112	0.6132	0.0010
5	0.6071	0.6070	0.0010
6	0.6044	0.6071	0.0010
7	0.6024	0.6024	0.0010
8	0.6006	0.6057	0.0010
9	0.5988	0.6028	0.0010
10	0.5978	0.6034	0.0010
11	0.5977	0.6038	0.0010
12	0.5967	0.5999	0.0010
13	0.5961	0.6008	0.0010
14	0.5955	0.6018	0.0010
15	0.5950	0.6003	0.0010
16	0.5942	0.6015	0.0010
17	0.5939	0.5999	0.0010
18	0.5934	0.5999	0.0010
19	0.5929	0.6022	0.0010
20	0.5924	0.5981	0.0010

Table A.1: Summary of Training log - Epochs 1 to 20

Epoch	Loss	Validation Loss	Learning Rate
21	0.5875	0.5955	0.0005
22	0.5870	0.5954	0.0005
23	0.5862	0.5955	0.0005
24	0.5858	0.5950	0.0005
25	0.5854	0.5957	0.0005
26	0.5847	0.5993	0.0005
27	0.5846	0.5942	0.0005
28	0.5843	0.5940	0.0005
29	0.5841	0.5952	0.0005
30	0.5832	0.5955	0.0005
31	0.5831	0.5936	0.0005
32	0.5826	0.5942	0.0005
33	0.5819	0.5965	0.0005
34	0.5813	0.5941	0.0005
35	0.5813	0.5961	0.0005
36	0.5805	0.5966	0.0005
37	0.5801	0.5937	0.0005
38	0.5798	0.5925	0.0005
39	0.5792	0.5934	0.0005
40	0.5789	0.5941	0.0005

Table A.2: Summary of Training log - Epochs 21 to 40

Epoch	Loss	Validation Loss	Learning Rate
41	0.5736	0.5908	0.0001
42	0.5730	0.5902	0.0001
43	0.5726	0.5904	0.0001
44	0.5724	0.5899	0.0001
45	0.5723	0.5898	0.0001
46	0.5721	0.5901	0.0001
47	0.5718	0.5900	0.0001
48	0.5718	0.5900	0.0001
49	0.5714	0.5900	0.0001
50	0.5713	0.5902	0.0001
51	0.5711	0.5900	0.0001
52	0.5710	0.5903	0.0001
53	0.5708	0.5897	0.0001
54	0.5706	0.5897	0.0001
55	0.5705	0.5895	0.0001
56	0.5702	0.5902	0.0001
57	0.5700	0.5900	0.0001
58	0.5698	0.5895	0.0001
59	0.5698	0.5894	0.0001
60	0.5696	0.5890	0.0001

Table A.3: Summary of Training log - Epochs 41 to 60

A.3 Gauss-Newton

The Gauss-Newton method is among the most fundamental optimization algorithms. It operates on the principle of performing a first-order Taylor expansion on:

$$f(x + \Delta\delta) \approx f(x) + J(x)\Delta\delta \tag{A.1}$$

Given the provided framework, the present objective is to locate the descent vector $\Delta\delta$ such that the norm $\|f(x) + \Delta\delta\|^2$ is minimized. This necessitates solving a linear least squares problem,

$$\Delta\delta^* = \arg\min_{\Delta x} \frac{1}{2}\|f(x) + J(x)\Delta\delta\|^2 \tag{A.2}$$

To determine an extremum, derive the aforementioned objective function with respect to $\Delta\delta$ and equate this derivative to zero.

$$\begin{aligned}\frac{1}{2}\|f(x) + J(x)\Delta\delta\|^2 &= \frac{1}{2}(f(x) + J(x)\Delta\delta)^T(f(x) + J(x)\Delta\delta) \\ &= \frac{1}{2}\left(\|f(x)\|^2 + 2f(x)^T J(x)\Delta\delta + \Delta\delta^T J(x)^T J(x)\Delta\delta\right)\end{aligned} \tag{A.3}$$

Taking the derivative of $\Delta\delta$,

$$2J(x)^T f(x) + 2J(x)^T J(x)\Delta\delta = 0 \tag{A.4}$$

rearrange the eq(A.3),

$$J(x)^T J(x)\Delta\delta = J(x)^T f(x) \tag{A.5}$$

This system is linear and is commonly termed the Gauss-Newton equation. Let's denote the coefficient on the left-hand side of the equation as H and the right-hand side as g. The equation then transforms as per this notation,

$$\mathbf{H}\Delta\delta = g \tag{A.6}$$

Recognizing the left side as $\mathbf{H}$ is not arbitrary. When compared to the Newton method, it's evident that Gauss-Newton employs $\mathbf{J}^T\mathbf{J}$ as an approximation to the second-order Hessian matrix in Newton's method, thereby obviating the need to compute $\mathbf{H}$.

Algorithm 1 Gauss-Newton Algorithm

```
Require: Initial value $x_0$
 1: $k \leftarrow 0$
 2: while True do
 3:     Compute the Jacobian matrix $J(x_k)$
 4:     Compute the error $f(x_k)$
 5:     Solve the increment equation: $H\Delta x_k = g$
 6:     if $\Delta x_k$ is sufficiently small then
 7:         break
 8:     else
 9:         $x_{k+1} = x_k + \Delta x_k$
10:         $k \leftarrow k + 1$
11: return $x_k$
```

While the Gauss-Newton method has its limitations, notably its lack of guarantee in finding a local minimum, it remains a significant technique in the realm of nonlinear optimization. Many algorithms, such as Levenberg-Marquad, within this domain can be traced back to variants of the Gauss-Newton method. Moreover, the optimization approach adopted in this research primarily relies on the Gauss-Newton method.