LEARNING DATA-DERIVED VEHICLE MOTION MODELS FOR USE IN LOCALISATION AND MAPPING

by

Kobie Wood Fick

Submitted in partial fulfillment of the requirements for the degree
Master of Engineering (Computer Engineering)

in the

Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology

UNIVERSITY OF PRETORIA

February 2018

SUMMARY

LEARNING DATA-DERIVED VEHICLE MOTION MODELS FOR USE IN LOCALISATION AND MAPPING

by

Kobie Wood Fick

Supervisor(s):	H. Grobler
Department:	Electrical, Electronic and Computer Engineering
University:	University of Pretoria
Degree:	Master of Engineering (Computer Engineering)
Keywords:	Neural networks, recurrent networks, tapped delay-line neural networks, data-derived non-analytical models, motion model, N^{th}-order recursive Bayesian estimator, SLAM, data-derived motion model evaluation

Various solutions to the Simultaneous Localisation and Mapping (SLAM) problem have been proposed over the last 20 years. In particular, extending the fundamental solution of the SLAM problem has attracted a great deal of attention. Most extensions address shortcomings such as data association, computational complexity and improving predictions of a vehicle's state. However, nearly all SLAM implementations still depend on analytical models to provide estimates for state transitions.

Learning data-derived non-analytical models for use during localisation and mapping provides an alternative that could significantly improve estimates and increase the flexibility of models. A methodology to learn motion models without knowledge of the higher-order dynamics is therefore proposed using tapped delay-line neural networks (TDL-NN). Incorporating the learned N^{th}-order Markov model into a recursive Bayesian estimator requires that the learned model be assumed independent of the transitional model, forming a black box estimator. Both real-world and simulated training data were evaluated, along with changes to the input data's format, to determine the best vehicle motion predictor.

Furthermore, an evaluation methodology is defined to asses how well the models could learn each motion type. A comprehensive analysis of the one-forward prediction using various statistical measures was used to determine the most appropriate metric. The methodology evaluated the predictions at different levels of depth, providing supplementary information on the type of motions that are learnable. Outcomes of the experiments revealed that inherently learning a vehicle's dynamics cannot be achieved using TDL-NNs. Currently the best that such an approach can learn is the delta between the vehicle's states. Consequently, modifications are required to the learning algorithms as well as the input data's format that will force the strategies to learn the higher-order dynamics.

LIST OF ABBREVIATIONS

AE_{max}	Maximum Absolute Error
AE_{median}	Median Absolute Error
$AE_{medianAD}$	Median Absolute Deviation of the Absolute Error
AE_{min}	Minimum Absolute Error
ANN-IBSS	Artificial Neural Network Iterative Bi-Section Shooting
ATE	Absolute Trajectory Error
CAT-SLAM	Continuous Appearance-based Tracking Simultaneous Localisation and Mapping
DOF	Degrees of Freedom
EKF	Extended Kalman Filter
ELC	Extreme Learning Machine
EM	Expectation Maximisation
FLANN	Fast Library for Approximate Nearest Neighbours
FPS	Frames per Second
FFNN	Feed-Forward Neural Network
GPs	Gaussian Processes
GPDM	Gaussian Process Dynamical Models
GPS	Global Positioning System
HFKM	Hierarchical Fuzzy K-means Clustering
HMM	Hidden Markov Model
IBSS	Iterative Bi-Section Shooting
ICC	Instantaneous Center of Curvature
IMU	Inertial Measurement Unit
IMCO	Inverse Motion Composition Operator
IMM	Interacting Multiple Model
INS	Inertial Navigation System
LDA	Linear Discriminant Analysis
LSE	Least Squared Error
LSTM	Long Short-Term Memory
MAD	Median Absolute Deviation
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error

MCO	Motion Composition Operator
MPEF-SLAM	Multi-sensor Point Estimate Fusion Simultaneous Localisation and Mapping
MSE	Mean Squared Error
NN	Neural Network
NeoSLAM	Neuro-Evolutionary Optimisation Simultaneous Localisation and Mapping
PF	Particle Filter
PSO	Particle Swarm Optimisation
RANSAC	Random Sample and Consensus
ReLU	Rectified Linear Unit
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
ROS	Robot Operating System
RPE	Relative Pose Error
RRR	Realising, Reversing and Recovering Algorithm
SIFT	Scale Invariant Feature Transform
SIS	Sequential Importance Sampling
Skew	Skewness
SLAM	Simultaneous Localisation and Mapping
Std. Dev	Standard Deviation
SURF	Speeded-Up Robust Features
TDL-NN	Tapped Delay-line Neural Network
UKF	Unscented Kalman Filter
URDF	Unified Robot Description Format
Var	Variance
Wi-Fi	Wireless Fidelity
XOR	Exclusive OR

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

1.1 PROBLEM STATEMENT

1.1.1 Context of the problem

Present-day robotic vehicles are increasingly required to accomplish complicated tasks. Usually these tasks aim to increase efficiency and safety of users through automation. Most, however, are single-purpose solutions that have limited functionality and considerable focus has therefore been placed on diversifying the tasks that a single robot can accomplish. Achieving general-purpose robotic systems requires that the systems exhibit intelligent behaviour during operation. In particular, spatial awareness is of the utmost importance for any system that needs to interact with its environment.

A fundamental requirement is therefore for a vehicle to observe and understand the surroundings as well as its placement within the environment at all times. The Simultaneous Localisation and Mapping (SLAM) problem refers to placing a vehicle in an unknown environment and procedurally generating a map while the vehicle is traversing the environment [1, 2, 3]. One of the first methods used to solve SLAM is based on recursive Bayes estimators, where the next vehicle and map state is predicted and updated according to environmental observations [4]. Commonly a vehicle's next state is predicted using motion models such as generalised odometry or velocity models [5].

Both models assume that a vehicle's state remain constant and that movement is directly related to vehicle control. For the odometry-based models the control is modelled as a change in each state variable, while velocity-based models use the current velocity vector. However, as with any generalised model, vehicle-specific motion remains largely unmodeled, leading to incorrect predictions. Correcting the predictions using landmarks and data-association techniques has allowed such systems to remove

these discrepancies during small-to-medium scale tests [3, 6, 7]. While the assumption does hold for small maps where the distances between loop-closures are relatively small, large loop traversals with many changes in trajectory cannot be as readily adjusted. The main cause of this shortcoming is drift in sensor measurements, which accumulate over large loops.

Graph solvers for the back-end of SLAM implementations have been used to improve the trajectory estimates [8, 9]. However, these methods have no verification metrics or methodologies that definitively prove that the approach improves motion estimates. Instead, graph solvers use the pose predictions to limit the motion to a maximum likelihood estimate.

An alternative method is using the vehicle's structure to improve predictions [10]. Commonly, characteristics such as the drive-type, kinematics and dynamics are included in the motion model to provide improved estimates [11, 12]. However, some motion models are highly non-linear and complex to solve analytically, thus requiring a thorough analysis of all vehicle and environmental parameters. In addition, many of the parameters are dynamic in nature, requiring constant revision as the vehicle traverses an environment. Lastly, there have also been approaches that learn some form of vehicle motion to improve vehicle predictions during SLAM [13, 14, 15].

1.1.2 Research gap

To form a more generalised model that encapsulates vehicle motion, machine learning techniques have been applied. Four methodologies exist to determine the vehicle models created during estimation: Classifying a vehicle according to drive-type or motion-mode [16, 17], learning to predict some of a vehicle's dynamic parameters to improve estimation [18, 19], learning the effect of control on a vehicle [20, 21, 15, 22] and learning an abstract representation of a vehicle's motion model [23, 14].

By classifying vehicles by type or motion mode, one is able to use the appropriate models to predict the next pose. However, classification approaches still rely on analytically defined models to predict the vehicle's next state. The second method, while simplifying some of the vehicle's calculations, still inherently depend on defining the vehicle's dynamic parameters, thus offering limited improvements. Learning the error of a motion model or the vehicle's plant model can help compensate for any errors that the model produces during estimation. As such, a model that represents a vehicle's motion model

isn't explicitly learned. Instead an estimator is learned that approximates the error that a change in vehicle state produces. Consequently the estimators are unable to provide predictions of what the vehicle's actual state should be.

The fourth method does not have any of the constraints of the other methods, as an abstract representation of the vehicle's motion model is learned. Thus learning the dynamics involved is not limited to those defined by the inputs. Similarly, any motion learned already contains motion compensation. However, most methods that learn an abstract representation of a vehicle's motion do not learn the full spectrum of vehicle dynamics. In addition, these methods are unable to provide accurate predictions outside of the training interval or the specific motion learned without biasing from analytical or plant models. Learning a fully abstract model that is able to encapsulate a vehicle's dynamics has therefore not yet been achieved. In addition, using such an abstract model in a SLAM context has yet to be demonstrated.

1.2 RESEARCH OBJECTIVE AND QUESTIONS

The proposed research aims to provide answers regarding motion model learning. Specifically, the research aims to provide answers regarding data-derived non-analytical models for use in a SLAM context. Hence the following research questions are addressed:

1. Is it possible to learn data-derived non-analytical motion models using neural networks?
    (a) Can *low-order* state data be used to encapsulate higher-order dynamics?
    (b) To what extent must the data's format be altered/filtered in order to provide suitable training data?
    (c) What impact does memory of the previous state data have on learning?
    (d) What are the types of motion that can be represented by the learned model?
2. How can one evaluate whether a motion model has been learned and what metrics should be used?
3. Can the data-derived non-analytical models provide an improvement over the corresponding analytical models in a SLAM context?

1.3 HYPOTHESIS AND APPROACH

A vehicle's motion can be described using its kinematic and dynamic characteristics. Kinematics are subject to a vehicle's dimensions, position, orientation and time. Any other variables represented within a kinematic setup are related to one or more of the aforementioned variables. By taking the dynamics of a vehicle and environment into account a fuller representation of the motion can be defined. As such, dynamic model takes mass/gravity, external forces and friction along with the vehicle's pose, dimensions and time into consideration.

Any analytical model is therefore limited to the higher-order dynamics and kinematics that have been taken into consideration. However, including additional variables leads to an increase in vehicle state and complexity that needs to be monitored. Furthermore, many of the dynamics involved in vehicle motion are unknown or approximated in order to make analytical solutions tractable and sequential.

A solution to the aforementioned problem is to define an abstract model that inherently captures the higher-order dynamics without explicitly defining what the dynamics are. The abstract model can then be trained by reducing the prediction error of the vehicle's next state. The challenge, however, is how to learn the higher-order dynamics given the data.

By noting that many of the higher-order dynamics are related to the low-order state variables as the difference changing over time, the dynamics can be encapsulated by including more previous states as input to the abstract model. Tapped delay-line neural networks (TDL-NN) offer an ideal mechanism through which the motion model can be trained, as a TDL-NN makes use of multiple previous states in order to predict the next state.

As the abstract model will be used in a SLAM context, a recursive Bayesian estimator is required to predict the vehicle's next state and update the prediction using some measurement model. The fundamental recursive Bayesian technique used in for SLAM is the Extended Kalman Filter (EKF)-SLAM algorithm. Thus the TDL-NN is incorporated into the EKF-SLAM algorithm to predict and update the vehicle's state.

1.4 RESEARCH GOALS

The primary aim of the research is to develop a methodology that is able to learn a vehicle's motion for use in a SLAM context. Specifically, the research aims to substitute the analytically defined models with a model derived from available data. Hence the learning approach followed should be able to learn any type of vehicle motion model. Determining the data type and format to learn the models therefore forms a significant portion of the research.

Furthermore, the learned model should be able to provide predictions for the vehicle's next state, given previously observed states. Thus, the learned model should be usable in practical applications such as recursive Bayesian estimators. An approach therefore needs to be defined to incorporate the learned model in a SLAM system. Lastly, the learned model should provide superior estimates of vehicle pose when compared to generalised analytical models. The learned model should also provide improved estimates irrespective of the measurement model selected.

1.5 RESEARCH CONTRIBUTION

The research conducted extended the current knowledge in three areas: Learning abstract motion models using historic state data, including the models within a recursive Bayesian estimator and an evaluation methodology to establish whether such a motion model could be learned. The first contribution provides answers on the required input data format, the neural network's (NN) activation functions and the amount of memory required. The frame-rates of real-world datasets were evaluated and a strategy was used to sub-sample the data without losing the majority of training data. In addition, the performance of the NNs were evaluated when vehicle control was included with the memory as input.

Incorporating data-derived models within recursive Bayesian estimators formed the second contribution. As recurrent NNs can be considered an N^{th}-order Markov process, incorporating the learned model into a first-order Markov process was addressed by defining the memory as internal to the NN. The recursive Bayesian estimator could therefore be considered conditionally independent of the NNs memory. A framework was created within the Robot Operating System (ROS) to test

the EKF-SLAM implementation and to evaluate the implementation's performance to other SLAM implementations.

Thirdly, a methodology was created to determine the accuracy of the learned models on different types of motion, with different controls and at different starting poses. The methodology employed a top-down approach whereby each individual trajectory was evaluated, followed by trajectories containing the same control velocities and finally, by motion type.

A last contribution of the research was a strategy to create simulated datasets using a differential drive vehicle's kinematic model. By training and evaluating the NNs' performance on single motion types, the learned models could be quantified by type. The primary advantage of the approach was that additional training data for each control and motion type could easily be generated. Consequently, the diversity required by the learning algorithm to reach usable solutions could also be quantified.

1.6 OVERVIEW OF STUDY

The research conducted provided a comprehensive study of the current literature, the proposed approach, evaluation methodology and experimental results. A summary of the available literature on SLAM, neural nets and model learning is provided in Chapter 2. In Chapter 3 the model learning strategy and SLAM implementation are discussed in detail. Chapter 4 provides an overview of the training data manipulation, evaluation methodology and metrics, as well as motivating the specific experiments conducted. Chapter 5 provides the results of the various experiments conducted and Chapter 6 discusses the observations, answers the research questions and reviews alternative machine learning strategies. Lastly, Chapter 7 provides a summary of the research conducted and discusses any alternative approaches and future work to extend and improve the findings.

CHAPTER 2 LITERATURE STUDY

2.1 CHAPTER OBJECTIVES

Various solutions to the SLAM problem have been implemented in recent years. Most extensions on the fundamental approaches address shortcomings such as data association, computational complexity and state estimates. Most SLAM implementations still depend on analytical models to provide estimates for state transitions and observations. Learning data-derived non-analytical models for use during localisation and mapping provides an alternative that could significantly improve estimates and increase the flexibility of models.

The following chapter therefore provides a comprehensive review of the literature concerning both SLAM and model learning. The SLAM algorithm's fundamental implementations are detailed in Section 2.2. Specifically, the EKF-SLAM and FastSLAM approaches are discussed in order to illustrate the challenges faced to reach a solution. Next, implementations that provide improvements to some of the deficiencies are detailed, along with any shortcomings that still needs to be addressed. Specifically, the problems associated with sensor drift and motion models are described.

In Section 2.3 the various model learning methodologies that have been used to learn dynamical models are investigated. Emphasis is placed on learning vehicle motion in order to predict the next state. The strategies are categorised according to the type of information that the model aims to learn. Lastly, Section 2.4 investigates the model learning strategies that have be applied in a SLAM context. The different machine learning algorithms are discussed, along with the methods employed to incorporate the learned motion models into SLAM.

2.2 SIMULTANEOUS LOCALISATION AND MAPPING

The key objective of SLAM is solving the problem of placing a mobile robot (or vehicle) in an unknown location, in an unidentified environment, and allow the vehicle to incrementally build a consistent map of the environment. The robot should also be able to ascertain its location within the map during traversal [1, 2, 3]. The theoretical problem of SLAM has been solved using a number of approaches in various domains. However, there are still a number of problems concerning the practical realisation of SLAM solutions [2].

The first general solution for SLAM emerged when probabilistic models were exploited to model vehicle motion and the environment. Notably, the work of Smith, Self and Cheeseman [1] played a crucial role in developing localisation and mapping techniques for mobile vehicles. In their paper, they described the use of uncertain spatial information to create a stochastic map of the environment [1]. The stochastic map tied together all the uncertain spatial information, and, in doing so, incrementally built a map as the information was obtained.

Creating such a map was achieved by using the movement of the robot along with the sensed objects (also called landmarks) that were in the robot's field of view. The solution was a two-step procedure that first estimated the movement using the odometry of a vehicle, followed by an update (or correction) step. The latter step compared new locations of the landmarks to previous observations to improve vehicle motion [1]. A key assumption for the SLAM solution was that a high degree of correlation existed between the landmark estimates and that these correlations would grow with successive observations. The implication was that in order to reach a full, consistent solution, the joint state of vehicle pose and landmark location was needed. In addition, every landmark and vehicle pose needed to be updated after each new observation.

Incorporating all the landmarks into the state vector, however, meant that the computational complexity scaled quadratically with the number of landmarks. Moreover, the approach assumed that the landmark errors would not converge and thus never exhibit steady-state behaviour [2]. With the realisation that the errors would converge and that the correlations between landmarks were crucial to finding convergent solutions, the first true solution to SLAM was created.

2.2.1 Fundamental SLAM approaches

Two main approaches are used to solve the SLAM problem. The first is commonly referred to as visual SLAM implementations [24, 25], while the second is known as GraphSLAM [26, 27]. The difference between the two is that visual SLAM implementations use visual landmarks to estimate vehicle pose as a robot moves through an environment (or vice versa), while GraphSLAM implementations create a graph where the nodes represent the poses and landmarks connected to each other. Another difference between the two approaches is that visual SLAM implementations update the vehicle pose and map during operation, while GraphSLAM approaches commonly exploit all the poses to estimate the full trajectory before creating the map.

As a result, GraphSLAM can more readily be employed in large-scale mapping applications, as the approach does not create maps before all the initial poses have been captured. A consequence of capturing all the available information before mapping is that GraphSLAM approaches mainly operate off-line. Visual SLAM, in comparison, uses odometry information to provide pose estimates, which are correlated to the map or landmarks to provide an estimate of the transform between frames. Note that both approaches adopts a state-space model to perform updates of the map and poses.

2.2.1.1 EKF-SLAM

The EKF-SLAM algorithm provides estimates of the vehicle's state transition using analytical models for the motion and environmental observations. Thus EKF-SLAM is very similar to the standard EKF algorithm [28] for tracking problems, where the EKF linearises a Gaussian distribution using a first order Taylor expansion. However, the EKF-SLAM algorithm has notable shortcomings, as listed below:

1. Convergence of the map is subject to a lower bound, determined by the initial uncertainty in the position of the robot.
2. As the correction step requires the joint covariance matrix of all the landmarks be updated, the computation required scales quadratically with the number of landmarks detected.
3. EKF-SLAM is sensitive to incorrectly associated landmarks and observations, especially the loop-closure problem where the robot returns to re-observe landmarks in the map.

4. As EKF-SLAM linearises around the local mean to handle non-linear models, the algorithm can run into substantial problems regarding consistency. Convergence and consistency can therefore only be guaranteed for linear models.

2.2.1.2 FastSLAM

The FastSLAM algorithm [29, 30] removed the linear Gaussian assumption of EKF-SLAM for state transition. Instead, the FastSLAM algorithm uses Monte Carlo sampling (a particle filter) [31, 32] to represent the non-Gaussian probability distribution. However, the state space becomes markedly high-dimensional, making the algorithm practically infeasible because of the amount of computation required. To overcome the computational limitations, Rao-Blackwellization [33] is applied to the state-space, which exploits the product rule of probability to partition the joint state space. The result is that only the marginal probability needs to be sampled, which significantly reduces the computation required.

Particle propagation for FAST-SLAM can be implemented in a number of ways. A frequently used method is sequential importance sampling (SIS) [33] which weighs all the samples in the distribution. Consequently, only samples with some minimum weight are propagated forward as the samples approximate the actual distribution. A known problem with SIS is that the variance of the samples grows as the approximation error increases. Re-sampling allows the weights to be returned to uniformity at the cost of losing the history of the particle's information. Thus SIS with re-sampling assumes that the system becomes increasingly independent of the process noise from the preceding states.

2.2.2 Improvements to SLAM

Many SLAM solutions are built upon the basis of the aforementioned approaches. Recent work to improve SLAM solutions include using better representation methods such as OctoMaps [34], improving vehicle odometry estimates using embodied information [10] and applying visual odometry to create dense 3D maps [35]. In addition, methods for handling dynamic environments [36], fusing sensor data [37] and increasing the robustness of loop-closures [38, 39] have allowed SLAM solutions to function in both indoor and outdoor environments. Most SLAM improvements can be categorised as:

1. Reducing the computational complexity.
2. Improving data association.
3. Extending environment representation.
4. Improving the accuracy of predictions.
5. Optimising the predicted trajectory (loop-closures).

The subsequent sections provide a brief summary of the improvements offered in each of aforementioned fields.

2.2.2.1 Computational complexity

As the SLAM problem scales quadratically with the number of landmarks in the map, computational complexity is one of the core problems that had to be addressed. To decrease computational complexity, methods such as linear-time state augmentation, sparsification of information, partitioning updates and sub-mapping the environments have been used [3].

A notable SLAM algorithm that reduces an EKF-SLAM implementation to $\mathcal{O}(n)$ was proposed by Perez et. al. [40]. Using a divide and conquer strategy to join local maps in a hierarchical fashion allowed the algorithm's computational complexity to scale linearly with an increase in map size. Furthermore, the approach did not approximate the state as many other $\mathcal{O}(n)$ EKF-SLAM approaches.

2.2.2.2 Data association

Data association is the ease with which a landmark is recognized at different time instances and how such landmarks are related to one another. Data association is particularly important when a vehicle returns to a previously mapped environment, as incorrect association can lead to catastrophic failures (also known as the loop-closure problem). Methods used to solve data association include batch-validation, multi-hypothesis techniques and appearance-based methods [3].

Much research in appearance-based SLAM approaches have focused on increasing the recall rates of previously visited locations while maintaining a very low number of false positives [6, 41, 39]. One of the most successful approaches to achieve the recall rates was using a visual "bag-of-words" to match

images to each other, as demonstrated by the fast appearance-based mapping (FAB-MAP) [6, 41] and continuous appearance-based tracking (CAT)-SLAM [39] algorithms.

The CAT-SLAM algorithm made use of appearance-based place recognition along with spatial filtering of geometric SLAM algorithms to create a continuous trajectory. In addition, loop-closure detection was more robust because the vehicle's pose was taken into consideration along with multiple subsequent observations of the environment. Furthermore, the algorithm was able to map the environment using either visual or odometry information, as well as a combination of the two.

The Realizing, Reversing and Recovering (RRR) algorithm [38], which is one of the more recent SLAM variants, pays particular attention to place recognition and how to handle incorrect decisions. The aim of RRR is to reduce the impact that any incorrect decision during loop-closures has on the environment model. A consensus-based approach was therefore implemented to detect and correct any faulty loop-closures. The approach used all the available sensor information to facilitate the best possible correction. Furthermore, the RRR algorithm is able to handle data from multiple sessions and incorporate new information into previously mapped areas that are spatially unconnected to create a larger map.

An alternative approach that has been used to increase the robustness of data association is data fusion. Strategies such as multi-sensor point estimate fusion SLAM (MPEF-SLAM) [37] fuses sensor information from a monocular camera and laser range finder. MPEF-SLAM fused the state variables and covariance estimates of a mono SLAM and laser SLAM implementation and back-propagated the fused values to each individual SLAM process. By employing two parallel SLAM algorithms the system increased robustness with regard to sensor failure. Furthermore, MPEF-SLAM demonstrated an improvement in localisation and a reduction in the state's covariance. However, the cost of these improvements is an increase in computational overhead that does not allow for real-time localisation and mapping.

An approach by Indelman et. al. fused sensor information by incorporating all the data into factor graphs [42]. Factor graphs, in essence, provide connectivity between variable nodes and sensor measurements. The advantage of using factor graphs is that new sensors are considered to be new factors and can therefore easily be incorporated into the graph. Similarly, if sensor measurements suddenly become inaccessible as a result of communication loss or faulty sensors, the system simply

removes the factors from the graph. A difficulty in using these factor graphs is that sensors operate at different frequencies. An inertial measurement unit (IMU), in particular, usually operates at a much higher rate than a global positioning system (GPS), range sensor or camera. Hence, an IMU pre-integration technique was used to reduce the frequency at which the measurements are taken.

Handling landmarks that change dynamically over time was one of the failure points of early SLAM implementations. The reason was that early SLAM implementations assumed that the environment would remain static during localisation and mapping. As a result, dynamic landmarks often had serious effects on the system during localisation.

An EKF-SLAM system that was able to handle dynamic changes in landmark locations [36] was recently implemented. A key assumption of the approach was that the dynamic landmarks were independent of static landmarks. Thus the system classified the landmarks as new, existing, ingoing or outgoing [36]. The new and existing landmarks were assumed to be static and the ingoing and outgoing landmarks dynamic. The ingoing landmarks represented landmarks that were detected in a different location from where the landmarks were previously observed, while outgoing landmarks were used when a landmark disappeared from previously the observed location.

An alternative approach by Luo et. al. preformed dynamic object detection using the concurrent outdoor SLAM and moving object tracking (SLAMMOT) algorithm [43]. The algorithm was derived from graph-based SLAM with dynamic object detection solved through a multi-sensor fusion strategy. To achieve recognition of the moving objects, two processes were required. The former fused the laser measurements with stereo vision cameras to establish if two objects were the same, while the latter associated moving objects with one another.

2.2.2.3 Environmental representations

Simple, discrete landmarks described by geometrically primitive structures are often unable to describe complex environments satisfactorily. Improving the representation of an environment in which a robot finds itself has consequently seen much improvement. Solutions such as delayed mapping, non-geometric landmarks and trajectory estimation [3] are some of the implementations that effect improvements to environmental representations.

Occupancy grids were one of the first representations adopted in autonomous robotics to model an environment [44]. As the name suggests, each map is represented in a grid format, of which each grid cell could only be in one of two states, occupied or unoccupied. Occupancy grids are commonly used in robot navigation tasks such as path planning and obstacle avoidance. A core limitation of occupancy grids is that inconsistent maps are created due to the high-dimensional environment space being decomposed into a one-dimensional estimation of occupancy [45].

Point-clouds [46] are used to model 3D environments and objects as simple points in 3D space. However, point-clouds do have a few well-known limitations. The first, and largest, limitation is that point-clouds cannot represent any type of volume, and thus no information can be gleaned about size or shape without first relating a point to neighbouring points. Point-clouds also regularly have missing data in the form of empty spaces, or "holes". The holes are not only due to point-cloud's lack of volume, but also to the resolution of the devices capturing the points as well as occluding objects. Better 3D representations have therefore been developed to overcome the point-clouds' limitations, which include implementations such as octrees [47] and OctoMaps [34].

Octrees [47] are a hierarchical data structure that describes and models large 3-D spaces as cubic volumes (commonly referred to as voxels). The tree structure represents each of the corners of a cube as a node in the tree. To obtain a full tree structure each parent node can be recursively subdivided into child nodes by using the corners. The procedure continues until the minimum voxel size is reached. The advantage of octrees is that the tree can be cut at any level to provide a coarser representation of the 3D space. Generally octrees are used to represent the occupancy of the cubes where occupancy corresponds to a Boolean value stating whether the node is occupied or not. However, the problem with octrees is that nodes in the tree exist that are uninitialised, which could represent both free and unknown space.

Newer 3D representation techniques, such as OctoMaps [34] provide methods to handle large 3D maps. OctoMaps improve upon point-clouds and octrees by creating discrete labels for each of the three states in which the cubic volumes could be. These states allow the OctoMap to specifically represent unmapped areas instead of just denoting the areas as free space. To handle dynamic environments OctoMaps define occupancy using a probabilistic approach that adaptively changes occupancy as observations occur.

2.2.2.4 State predictions

Drift in sensor measurements over time is still one of the main failure points of SLAM [48]. As the drift becomes larger, estimates for the movement model become less coherent and hence limit the size of the map that can be generated. To reduce sensor drift, methods such as bundle adjustment [49] have been used. Bundle adjustment seeks to produce the optimal 3D structure by estimating the best possible parameters of the system. To achieve the optimal parameters requires minimizing a cost function associated with the model. An alternative approach that has recently been used to reduce odometry errors was to periodically reinitialise the system using "keyframes" [48]. Specifically, the keyframes were used to relocalise the motion of a vehicle by using every second frame to estimate the trajectory and account for drift reduction. Once estimated, the remaining frames are corrected by the frames before and after them.

Embodied SLAM implementations exploited the embodied information inherent in a robot to improve localization and mapping [10]. The advantage of incorporating embodied information into the SLAM problem is that the interactions of a robot with the environment can be taken into account. This SLAM implementation not only uses visual-visual and embodied-embodied associations, but also visual-embodied information to improve localisation and navigation through a Rao-Blackwellized particle filter. Furthermore, the implementation combines a pose-graph-based SLAM implementation with particle filter SLAM to perform explicit loop-closing.

The Closed-form Online Pose-Chain (COP)-SLAM [50] algorithm aims to decrease computational complexity by sparsifying a pose-graph into a pose-chain. An important property of pose-chains is that each node has two distinct edges. The first represents successive edges, while the second represents loop-closures to connect the current node to previous nodes. The pose-chains are optimized using trajectory bending to provide relative transformations until an absolute transformation is reached. The implication is that when a loop-closure encompasses the previous loop, the weights of the loop's transformation are lower than the weights not in the previous loop, making COP-SLAM piecewise optimal with regard to loop-closures.

There are also a number of approaches that have taken vehicle motion into consideration with respect to SLAM. A vehicle's motion model was exploited within a MonoSLAM implementation to provide relative pose constraints in [12]. MonoSLAM implementations are highly dependent on the number

of features maintained within a map and consequently often have large computational overheads. By calculating the relative pose constraints from the image points, vehicle motion could be limited. The implementation differed from previous approaches by employing vehicle models with non-holonomic constraints [12] (such as differential drives, skid steer models and Ackermann steering) instead of a generalised odometry or velocity models. Thus the vehicle did not have free motion in each dimension. One constraint experienced with the approach was that only 2D motion was estimated, thus limiting the estimates to planar motion.

A similar approach was followed in [51] that used a single-track model. The single-track or bicycle model is often used to simplify Ackermann steering models by reducing the number of wheels to one front and one rear wheel. The single-track model was utilised within a local BA framework [49] to improve pose estimates of the vehicles. To calculate the pose between frames a 1-point RANSAC approach was followed, as described in [7]. In addition, the vehicle's dynamical properties were taken into consideration in order to account for vehicle side-slip. The experimental results obtained tested both high and low velocity datasets. Their results indicated that using visual features, in-vehicle sensors and a vehicle motion model provided the closest pose estimate to ground truth for the high-speed as well as low-speed datasets.

Incorporating the dynamics of a vehicle into prediction models to improve dead-reckoning was also proposed. One such approach investigated the effects of including the dynamics, such as wheel slippage, during motion for a skid-steer drive system [11, 52]. Specifically, the approach determined whether the inclusion of the geometrical relationships between the wheels' Instantaneous Center of Curvature (ICC) and the vehicle velocity could improve pose estimates. The approach demonstrated how to experimentally obtain a bounded ICC [11] within the motion plane using genetic algorithms [53]. The results showed a marked improvement in dead-reckoning when compared to the default symmetric model.

Making use of multiple motion models to provide estimates during localisation extended some of the previous work [54]. As different motion models performed better under certain conditions than others, the authors proposed a framework to incorporate the models into SLAM by fusing multiple recursive Bayesian estimators. Specifically, an Interacting Multiple Model (IMM) [55] was created that fused the dynamic and kinematic models for a single-track vehicle. The kinematic model was used to represent low-speed, low-slip conditions, while the dynamic model was used to represent high-speed, high-slip

conditions. All the tests were conducted in the CarSim simulated environment, with the IMM filter yielding fewer errors than each individual model over various driving conditions.

In [56] the IMM filter approach was extended to include sensor fusion for onboard inertial navigation systems (INS) and a GPS. Similar simulations were implemented as in [55], along with practical experiments with a car. The results demonstrated that dynamic models were less error-prone during high-speed simulations, while kinematic models provided better accuracy during low-speed simulations. Furthermore, the tests demonstrated that the IMM filter outperformed both individual filters.

2.2.2.5 State optimisation

Motion compensation refers to the correction of initial pose/trajectory estimates and involves exploiting current knowledge of the vehicle's pose to correct previous estimates to ensure consistency throughout the localisation and mapping process. Motion compensation can be introduced using three distinct methods. The first method that most SLAM implementations make use of is direct compensation through filtering algorithms such as the EKF and particle filters [33]. Thus the estimate is implicitly compensated through the limits defined within the filtering algorithm. A second technique that can be used is a consensus-based approach [7, 57] that fuses multiple odometry measurements from various sources.

Loop-closure is the third form of motion compensation that most SLAM implementations apply. As detected loop-closures have fixed start- and end-points, a vehicle's trajectory can be limited to certain upper bounds [58]. A drawback to loop-closure rectification is that the limits are related to the length of the loop. Thus as the loop's length increases, the corrections become less accurate. The loss is particularly observable in the middle of the loop, where most of the uncertainty lies [58].

In order to correct the aforementioned errors, back-end pose optimisation algorithms were used to find a maximum likelihood solution for the trajectory [9]. Commonly, the optimisation algorithms operates after a loop-closure has been detected to constrain the vehicle's trajectory further. Many graph-based SLAM approaches include an optimisation step to compensate any pose inconsistencies. In particular, focus is placed on the inconsistencies between the pairwise pose estimates of pose-graphs. Non-linear

optimisation methods such as Gauss-Seidel relaxation, Gauss-Newton and Levenberg-Marquart solvers [59] are regularly used to correct the pose estimates.

The g^2o graph-solver improved upon the aforementioned methods by exploiting the structure and sparsity of graphs [9]. This included removing the assumption that all the parameters were part of an Euclidean space and representing the parameters in an alternate space from the mean increments. An alternative to g^2o is the Tree-based netwORk Optimizer (TORO) [8] algorithm that makes use of maximum likelihood with stochastic gradient descent to ascertain the most likely estimates of the nodes in the graph. Furthermore, the nodes are parametrised using a tree structure where the difference in pose is used between two nodes.

2.3 MODEL LEARNING

Recently there have been a number of implementations that learn some form of a vehicle motion model. Popular among these methods are models that provide predictions for robotic arms as well as vehicle motion. Generally the learning algorithms can be categorised into four types: Classification, parameter estimation, control estimation and model estimation.

Classification focuses on identifying the type of motion detected and is used to identify the type of vehicle (such as a car, motorcycle, truck, omni-steer, etc.) or type of motion (straight-line, rotation, gradual turn, etc.). Parameter estimation, in comparison, learns to estimate a dynamical system's parameters in order to improve accuracy.

Control and model estimation are frequently used interchangeably in a model learning context. However, a distinction does exist between the two. Learning control can be equated to learning a plant model, which in the case of no input, provides output identical to the input. Thus vehicle control can learn the changes in state given control input. Model estimation, on the other hand, refers to learning a complete model. Hence model estimation is able to provide estimates of the new state given the previous state (which may include control). The subsequent sections describe some of the implementations employed to achieve model learning.

2.3.1 Classification

By identifying the type of vehicle or motion observed, the correct analytical model can be used to make predictions. A vehicle classification strategy was implemented in [16] that categorised the vehicles and corresponding motion paths in order to perform traffic flow analysis on highways. The approach used a vehicle's track patterns from live video to classify eight different types of vehicles. Furthermore, blob features were transformed using linear discriminant analysis (LDA) [60] and a weighed k-nearest neighbour classifier in order to ease identification.

Once detected, the vehicle was tracked through a standard Kalman filter. Using the aforementioned approach, both flow and behaviour analysis was used to determine the vehicle's speed and predicted path, as well as to detect anomalies in the environment. The experimental outcomes of [16] demonstrated that vehicle speed and path could regularly be estimated. However, the system failed when vehicle occlusion or irregular motion was observed.

Identifying a vehicle's motion-mode energy was proposed by [17]. The purpose of the research was to control the vehicle's suspension depending on the mode in which the vehicle found itself. The motion-modes were identified by a NN using the suspension deflections as input. Three motion-modes needed to be identified by the NNs: Ground excitation, a fishhook manoeuvre (steering wheel input) and combined braking and steering operations. Once identified, the correct strategy could be applied for vehicle handling. The approach compared the NN's performance to a motion-mode energy method and demonstrated that the correct motion-mode could be identified under various conditions.

2.3.2 Parameter estimation

A number of approaches have learned some of the characteristics of vehicle models. One notable approach is that of Sun, Wang, Er and Liu [19], which applied extreme learning machines (ELC) [61] to track surface vehicles with unknown dynamics. The general idea of an ELC is to randomly generate hidden nodes in a single feed-forward NN [62] and analytically determine the weights using a pseudo-inverse technique. The main advantage ELCs is that no iterations are required over the data to resolve the hidden node parameters, as randomness is embedded into the ELC's regressors.

The authors of [19] used such a network to determine unknown dynamics and external disturbances of 3-DOF (degrees of freedom) surface vehicles (i.e. boats, ships, etc.) by adopting a sliding surface of tracking errors and first derivatives. Consequently, unknown dynamics and uncertainties were encapsulated by a non-linear function and used as system states. The result was an ELC scheme that did not require any prior information of the dynamics for vehicle tracking and approximation. Simulation results of the scheme revealed that the NN could estimate the dynamics of the system within 20-25 seconds.

Learning the dynamic parameters for unmanned aerial vehicles was investigated in [18] by incorporating NNs with an Iterative Bi-Section Shooting (IBSS) methodology. In their work, NNs were trained to estimate a subset of the parameters in order to ease the computational load. Once estimated, the system continued to bisect the remaining parameters until the estimates reached a predefined minimum. The resulting structure required less bi-sectioning loops to reach convergence. Furthermore the NN-IBSS method consistently provided errors that were half an order of magnitude less during parameter estimation when compared to individual IBSS and NN methods.

2.3.3 Control

Learning how control parameters affect dynamical systems is one of the first model learning strategies followed. One of the first instances of learning control was by learning plant models using NNs [20]. Both multi-layer and recurrent networks with various configurations were used to model the non-linear dynamics inherent in plant models. A core assumptions of the approach was that the outputs were bounded within a certain interval. The experimental results demonstrated that non-linear control could be learned by using the correct configuration and learning rates.

Batch vs. online learning for non-linear control was evaluated by [63] using both multilayer and feed-forward networks. Most of the work was based on [20], with the focus on establishing what strategy best suited learning plant models for control. Online training was demonstrated to reach similar error rates to batch training, with online training providing a more stable decrease in error over the epochs.

Learning the control of robotic arms with NNs was investigated by [13, 64]. In their work, a sliding motion on the surface was applied in order to adaptively control the arm for different payloads. While the results demonstrated that control could be learned, both contained significant errors during initial control/start-up. The strategy was further extended by [65], where two distinct NNs were trained, one for equivalent control and one for corrective control. Although the work demonstrated that the strategy could smooth arm control, a direct comparison of the arm's accuracy was not provided.

An adaptive strategy for inverse control of dynamic systems was created using NNs [66]. The approach divided adaptive filtering into three main components: Identifying the dynamical system, controlling the system's response and a disturbance canceller to minimise output disturbances. The dynamical system was modelled using a feed-forward NN, while the disturbance canceller was modelled using a TDL. Hence a recurrent NN was formed that re-incorporated the previous outputs into the system. While system identification could be learned almost perfectly, the feed-forward tests revealed that the system could only provide correct predictions by including a copy of the controller to estimate the disturbances. Thus the algorithm was still dependent on the actual plant during execution.

An adaptive controller that compensated for wheel-slip and external forces was presented by [22]. The controller made use of NNs to provide estimates of wheel slip of a differential drive vehicle. Online training was implemented by assuming that the error was bounded by the desired trajectory and actual trajectory output from the adaptive controller. Thus the NN could account for any erroneous forces acting on the vehicle by adapting the weights. Simulations conducted in Matlab demonstrated that the adaptive controller quickly stabilised to account for the errors while still updating the weights when necessary. However, practical experiments still need to be conducted to verify the improvements.

2.3.4 Model estimation

An approach to incrementally learn motion patterns and predict future motion was proposed in [14, 67]. The implementation allowed motion models to be learned in parallel with the predictions by using Hidden Markov Models (HMM) that can be grown as the observations increase (also known as Growing HMM). The implementation employed a topological map that had discrete state-spaces between state transitions, allowing a transition only if the states were neighbours. The topological map would then be updated after every observation while keeping the number and distance between

nodes to a minimum. Both simulated and real-world experiments were conducted in [14, 67], using visual tracking of vehicles in a parking environment. The approach was compared to an expectation maximisation (EM) and hierarchical fuzzy K-means (HFKM) clustering algorithms and shown to provide less prediction errors over 200 tested trajectories. Additionally, the actual model size of the Growing HMMs were significantly less than both the EM and HFKM algorithms.

2.3.5 Other types of learning

Bootstrapping consists of designing agents that learn models without any prior knowledge to achieve useful tasks. Using sensorimotor cascades (i.e. the actuators, external world and sensors), models for robots were learned using such an approach [68]. The primary focus was on designing agents that could learn any robot's dynamics or sensor model where the observations were high-dimensional and the dynamics non-linear. These sensori cascades were learned using diffeomorphisms (a "nice" smooth invertible mapping) of the sensory elements' space and used to provide long-term predictions.

The learned agents' prediction were extended to include motion planning in [69]. To accomplish learning, each action of a diffeomorphism needed to the associated in the observation domain. A generic search algorithm was implemented based on graph searching techniques. Furthermore, as node expansion is costly, a method to identify a redundant motion plan was implemented along with a method to pre-compute composite actions.

The bootstrapping algorithm was expanded to bilinear models for simple vehicles [70]. The vehicle models, which were an idealisation of mobile robots, were considered along with a set of sensors. The sensori dynamics were approximated by non-linear systems that assumed an instantaneous bilinear relation among the observations, commands and changes in observations. The robot was therefore described by the body (actuators and sensors) and the environment, while the vehicle itself was described as the vehicle body (the kinematics and sensors) and the environment. The outcome of the work demonstrated that the vehicles were able to perform servoing (or homing) using individual sensors.

2.4 MODEL LEARNING IN SLAM

Model learning approaches have also been applied in a SLAM context. The following section discusses some of the most notable approaches that have been successfully used within a localisation and mapping framework.

2.4.1 Gaussian processes

There have been a number of approaches that exploited Gaussian processes (GPs) to learn model dynamics. Notably the work of [71, 72] used GP dynamical models (GPDM) to learn the non-parametric models for human motion. In the aforementioned approaches each human pose was defined by a number of Euler angles for various joints along with the torso and translational velocities. A motion capture system was used to collect training data for each of the motions. The actual learning is based on a GP latent variable model [73] to reduce the latent positions into a generative model for the poses. Hence training the GPDM requires estimating the latent positions as well as kernel hyper-parameters. Once learned, the GPDM could be used to track people by extracting the locations of joints in image data.

GPs have also been employed to learn transitional and observational models in Bayesian filtering applications [23, 74]. The Bayesian filters utilised consisted of an EKF, unscented Kalman filter (UKF) and a particle filter. The GP was trained using the ground truth states as output, with the state transitions (deltas) and control as input. An application to track a robotic blimp using the trained GPs was demonstrated to show that the GPs could be sufficiently trained. One caveat of the approach in [23, 74] was that the GPs could not provide accurate estimates for test data whose distance was not within the training data's limits. To overcome the problem of a narrow prediction interval, a parametric model was included in the transitional and observational models (also called an enhanced-GP process). The algorithm therefore depended on both the learned GP model as well as an analytically defined model to provide predictions.

Other GP approaches that have been used in a SLAM framework consist of providing pose estimates given magnetic sensor readings [75], [76] and Wi-Fi-based sensor readings [77, 78]. In [76] and [75] a Rao-Blackwellized particle filter was used to provide pose estimates, while a GP was employed to

model the magnetic field map. The magnetic field vector was modelled by three independent GPs to generalise the multivariate normal distributions into metric sensing space. The measurements were then used to correct the odometry information by using the deviation in the magnetic fields. However, the method relied heavily on loop closures to provide proper compensation.

Wi-Fi-SLAM [77], in comparison, made use of GPs to provide pose predictions by exploiting Wi-Fi signals. The approach used the signal strength to determine the latent variables (2D xy-coordinates) of the device and assumed that the locations and signal strength were highly correlated. The implication is that similar signal strength observations would be observed at similar locations and that locations near one another should observe similar signal strengths. Furthermore, the signal strength maps had to be built without relying on location data, thus treating the locations as hidden/latent. In [78] the computational complexity of Wi-Fi SLAM was improved by using a GraphSLAM approach.

Lastly, estimating a continuous-time non-linear batch state using a non-parametric model was addressed in [79]. By combining a GP with Gauss-Newton optimisation the scarcity of measurements was addressed while simultaneously providing a transitional model dependent on the physical properties of the system. Both simulation and experimental results demonstrated an improvement over the Gauss-Newton approach when using the system in a SLAM context.

2.4.2 Neural networks

There have also been various approaches to incorporate NNs into the SLAM. One such approach was implemented in [21], where a NN was combined with an EKF to improve the pose estimates. The approach used a NN as a plant for the EKF in order to model the non-linear errors, as an EKF assumes white Gaussian noise [80]. Hence the NN augmented the EKF state predictions by modelling the errors inherent in the transitional model. An Ackerman steering system traversing a 100m $\times$ 100m map was used to test the validity of the system [21]. The results showed no significant improvement with an unbiased model. However, there was some improvement in bounding the errors over a standard EKF when a biased model was used. Furthermore, the biasing was based on known vehicle parameters and assumptions and would therefore not necessarily hold for other dimensions or vehicle models.

An improvement on the work in [21] was suggested by [81]. The approach differed from the previous

implementation by changing the input vector of the NN to only operate on the control command instead of the robot's pose. In addition, the NN's structure was altered and the activation function changed to a hyperbolic tangent. The NN could therefore model the velocity error correction vector to compensate for the vehicle's velocity. A clear improvement over a standard EKF when using a differential drive robot was demonstrated using this approach.

The NN odometry error estimator in [21, 81] was improved by using a FastSLAM implementation instead of EKF-SLAM [82]. Simulations based on an Ackermann steering model and the DLR-Spatial Cognition dataset [83] were used as verification of the improvement over a Fast-SLAM 2.0 implementation. Another variant on the previous approaches is using a recurrent NN based on an Elman network [84] to improve the state estimates [85]. The results demonstrated an improvement over a general EKF-SLAM implementation as well as the NN-EKF implementation detailed in [21].

Radial basis function NNs have also been used for trajectory tracking in robotic manipulators [86]. Specifically, the approach was used in cleaning and detecting robots for control using sliding modes. A three hidden layer NN was used where the weights were updated using Lyapunov's stability theorem. This allowed for a robust, adaptive controller whose errors converged to a specified precision.

Adaptively adjusting the error covariance in an EKF by using NNs was investigated by [87, 88]. The NNs were trained to adaptively adjust the amount of noise within both the transitional and measurement models based on the amount of error between the theoretical and actual covariance in the innovation process. When such an error was sufficiently large the NN adjusted a scaling factor for both noise sources in order to eliminate the discrepancy.

Recently a new SLAM approach called Neuro-Evolutionary Optimization SLAM (NeoSLAM) was published that employed NNs with evolutionary programming to learn the motion error [15]. To train the NN, the input vector was defined as the difference between two consecutive poses of the robot along with the feature positions at each pose, while the output vector was the odometric error of the robot. As real-world experiments do not have ground truth, an evolutionary optimization algorithm was applied to evolve the weights using a cost function to estimate the ground truth [15]. Thus the NeoSLAM algorithm was able to *implicitly* model the robot kinematics and sensor characteristics. From their experiments, NeoSLAM outperformed both EKF-SLAM and FastSLAM [29].

2.5 SUMMARY

While numerous approaches have been aimed at improving vehicle motion and the corresponding predictions, most only aim to reduce the error of the prediction or by identifying the exact mode of operation. Approaches that focus on learning the motion model's dynamics require some bias or additional plant model to provide predictions outside the training interval. As such, the models that were learned do not explicitly encapsulate the entire problem spectrum and can therefore not be considered a fully learned motion model.

The aim of the subsequent research is to determine if a full motion model can be learned and incorporated into a SLAM system. Specifically, focus will be placed on learning models using NNs. The approach differs from the other NN implementations (such as NeoSLAM [15]) by using the vehicle's current state to predict the next state instead of the odometric error. Furthermore, instead of using knowledge of the vehicle, such as the Euler angles for various joints and translational velocities [71], memory of the state is used to learn the motion dynamics.

In addition, the research takes data formatting into consideration, including preprocessing steps required to provide usable training data, the amount of memory required to learn the vehicle's dynamics and the creation of simulated datasets to cover the problem space. Lastly, an evaluation methodology to determine the models' performance and types of motion that could be learned is described.

CHAPTER 3 APPROACH

3.1 CHAPTER OVERVIEW

The following chapter provides an in-depth description of the model learning approach adopted in this work to learn a vehicle's motion model. In addition, an approach is defined to incorporate the learned model into a recursive Bayesian estimator for use in SLAM. Using the general motion kinematics and dynamics, the variables of interest (state) are identified in Section 3.2. The higher-order dynamics are related to the base state variables and described as the change over the variables through time. Thus the model learning strategy is detailed in Section 3.3 by using a number of previous base variables in order to model the higher-order dynamics.

Section 3.4 describes the learning strategy using TDL-NN [66] with two types of inputs. The first considers using only the vehicle's state memory to learn the motion model, while the second incorporates vehicle control with the memory. Parameters that affect learning, such as momentum, annealing and activation functions, are discussed and the parameter's impact on robustness explained.

Incorporating the learned model into a SLAM algorithm is detailed in Section 3.5 after the two learning strategies have been defined. The model's predictions are analysed from a statistical viewpoint and related to an n^{th} order Markov model. The assumptions used to create a first-order Markov model are stipulated so that the model can be incorporated into an EKF-SLAM system. A full description of the SLAM algorithm with 3D landmarks is subsequently provided and related to the learned model. Figure 3.1 provides an overview of the general approach employed.

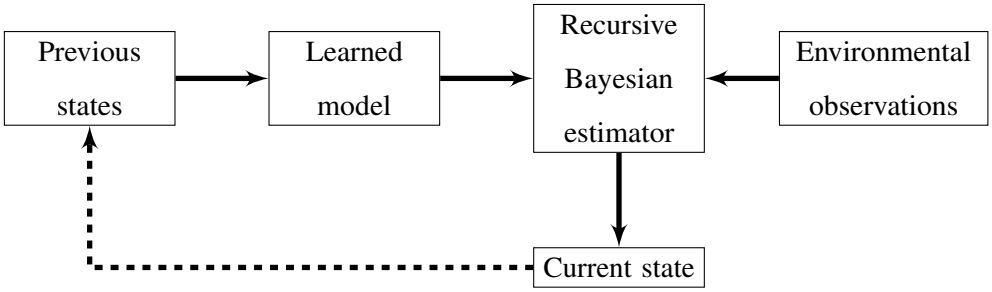


Figure 3.1. Overview of the procedure to incorporate a learned model into a recursive Bayesian estimator. The previous vehicle states (memory) is used by the learned model to provide a prediction. The recursive Bayesian estimator fuses the prediction with the environmental observations (landmark locations, etc.) to produce the current vehicle state. The current state then becomes part of the previous states and the estimation procedure continues.

3.2 MOTION KINEMATICS AND DYNAMICS

Any vehicle's motion can be described by taking the kinematic or dynamic characteristics into consideration. The kinematics are subject to the vehicle's dimensions, position, orientation and time. Any other variables represented in a kinematic setup are related to either one or more of the aforementioned variables. A vehicle's dynamics, in comparison, uses additional variables to provide a more accurate representation of the vehicle's motion. Specifically, dynamic equations incorporate a vehicle's mass, external forces, pose, dimensions and time into the model. Other variables such as friction are also often included in higher-order dynamic equations.

The following section describes two of the most commonly used motion models employed in a SLAM framework. The first is an odometry-based model that is directly linked to a vehicle's odometry control, while the second is a kinematic model of a differential drive vehicle. Both can be used as a baseline for comparison when evaluating the performance of a learned motion model during SLAM. For more information regarding the dynamic equations for a differential drive the reader is referred to Addendum C.

3.2.1 Odometry motion model

An odometry-based motion model defines the vehicle's state ($\mathbf{x}_k$) as the previous state that received some control input [5]. Consequently, if no control input is provided to the analytically defined model, the state will remain the same (see (3.1)). For odometry-based models the control input ($\mathbf{u}_k$) is defined as the change in a vehicle's position and orientation between subsequent measurements. As such, the transitional model for an odometry-based model can be defined as

$$\begin{aligned}\mathbf{x}_k &= f(\mathbf{x}_{k-1}, \mathbf{u}_k) \\ &= \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \cos(u_\theta) & -\sin(u_\theta) & 0 \\ \sin(u_\theta) & \cos(u_\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_\theta \end{bmatrix},\end{aligned} \tag{3.1}$$

where:

- x_{k-1} and y_{k-1} is the vehicle's previous position in 2 dimensions,
- θ_{k-1} is the vehicle's previous orientation (yaw),
- u_x and u_y is the change in the vehicle's position,
- u_θ is the change in the vehicle's orientation.

As any control applied to a vehicle model contains a certain amount of noise, a probabilistic definition of motion is needed. Specifically the probability $p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k)$ needs to be determined for each prediction. Incorporating the predictions into a recursive Bayesian estimator such as an EKF requires linearisation of the non-linear function through first-order Taylor expansion. Consequently, the Jacobians for both the state (3.2) and control (3.3) were calculated. The previous state's variables are therefore independent of each other, while the control is highly dependent on the change in yaw.

$$\mathbf{A} = \frac{\partial f(\mathbf{x}_k, \mathbf{u}_k)}{\partial \mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

$$\begin{aligned} \mathbf{U} &= \frac{\partial f(\mathbf{x}_k, \mathbf{u}_k)}{\partial \mathbf{u}} \\ &= \begin{bmatrix} \cos(u_\theta) & -\sin(u_\theta) & -u_x \sin u_\theta - u_y \cos(u_\theta) \\ \sin(u_\theta) & \cos(u_\theta) & u_x \sin u_\theta - u_y \cos(u_\theta) \\ 0 & 0 & 1 \end{bmatrix}. \end{aligned} \tag{3.3}$$

3.2.2 Differential drive kinematics

The following section details the general kinematic motion for a differential drive vehicle [89]. As stated previously, kinematic motion depends on a vehicle's dimensions, pose and control. For a differential drive vehicle, the kinematics can be related to the vehicle's instantaneous center of curvature (ICC) and each wheel's velocity (v_r and v_l). In turn, these variables depend on the base-length of the vehicle (l_b) , the signed distance from the ICC to the vehicle's center (R), the heading angle (θ) and the rotational velocity (ω). Figure 3.2 provides the general kinematic setup for a differential drive vehicle.

From the dependencies, one can define the rotational velocity (ω), forward velocity (v_f) and signed distance from the ICC to the vehicle's center (R) as a function of each wheel's velocity and the base-length as

$$\omega = \frac{v_r - v_l}{l_b}, \tag{3.4}$$

$$v_f = \frac{v_l + v_r}{2}, \tag{3.5}$$

$$R = \frac{l_b(v_l + v_r)}{2(v_r - v_l)}. \tag{3.6}$$

Using the aforementioned results, each wheel velocity is therefore calculated as:

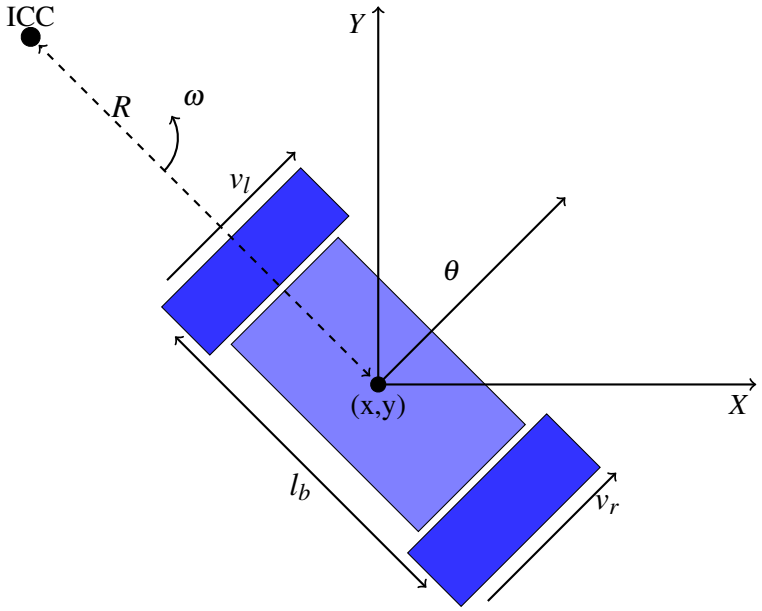


Figure 3.2. Differential drive kinematic setup using world coordinates as reference (x, y). The vehicle's characteristics depend on the base-length (l_b), instantaneous center of curvature (ICC) and each wheel's velocity (v_r and v_l). In turn, these variables depend on the signed distance from the ICC to the vehicle's center (R), the heading angle (θ) and the rotational velocity (ω).

$$v_r = \omega \left(R + \frac{l_b}{2} \right), \tag{3.7}$$

$$v_l = \omega \left(R - \frac{l_b}{2} \right). \tag{3.8}$$

From the equations, the ICC of a differential drive vehicle can be calculated. By using the vehicle's current pose and and the signed distance to the ICC, the exact location of the ICC is defined as:

$$ICC = \begin{bmatrix} x_k - R\sin\theta \\ y_k + R\cos\theta \end{bmatrix}. \tag{3.9}$$

A vehicle's current pose can therefore be defined in terms of the previous pose, the ICC, rotational velocity and time increment. As such, the general kinematic equations for a differential drive are [89]:

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} \cos(\omega_k \Delta t_k) & -\sin(\omega_k \Delta t_k) & 0 \\ \sin(\omega_k \Delta t_k) & \cos(\omega_k \Delta t_k) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} - ICC_x \\ y_{k-1} - ICC_y \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega_k \Delta t_k \end{bmatrix}. \tag{3.10}$$

Using (3.10), the probability model $p(\mathbf{x}_k|\mathbf{x}_{k-1}, \mathbf{u}_k)$ can be defined. For a differential drive's kinematic model the control input is defined as the left and right wheel velocities and the change in time. However, many datasets provide control in terms of the forward and rotational velocity instead of the wheel velocities. Thus, by substituting the forward velocity and rotational velocity into (3.6), the kinematics can be redefined as shown in (3.11).

$$f(\mathbf{x}_k, \mathbf{u}_k) = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + \begin{bmatrix} \cos(\omega\Delta t) & -\sin(\omega\Delta t) & 0 \\ \sin(\omega\Delta t) & \cos(\omega\Delta t) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{v_f}{\omega}\sin\theta_{k-1} \\ -\frac{v_f}{\omega}\cos\theta_{k-1} \\ \omega\Delta t \end{bmatrix} + \begin{bmatrix} -\frac{v_f}{\omega}\sin\theta_{k-1} \\ \frac{v_f}{\omega}\cos\theta_{k-1} \\ 0 \end{bmatrix}. \tag{3.11}$$

As with the odometry model, incorporating the model into a recursive Bayesian estimator requires linearisation. Hence the Jacobians for both the state **A** (3.13) and control **U** (3.12) variables were calculated. Observe from the equations that both the state and control variables are now dependent upon each other. Note that for readability (3.12) is written as a 9×1 vector instead of a 3×3 matrix and the subscript k is removed from all the variables.

$$\begin{aligned} \mathbf{U} &= \frac{\partial f(\mathbf{x}_k, \mathbf{u}_k)}{\partial \mathbf{u}} \\ &= \begin{bmatrix} \sin(\theta)\left(\frac{\cos(\omega\Delta t)}{\omega} - \frac{1.0}{\omega}\right) + \frac{\sin(\omega\Delta t)\cos(\theta)}{\omega} \\ \frac{v_f\cos(\theta)}{\omega}\left(\Delta t\cos(\omega\Delta t) - \frac{\sin(\omega\Delta t)}{\omega}\right) + \frac{v_f\sin(\theta)}{\omega}\left(\frac{1.0}{\omega} - \Delta t\sin(\omega\Delta t) - \frac{\Delta t\cos(\omega\Delta t)}{\omega}\right) \\ -v_f\sin(\theta)\sin(\omega\Delta t) + v_f\cos(\theta)\cos(\omega\Delta t) \\ \frac{\cos(\theta)}{\omega}(1-\cos(\omega\Delta t)) + \frac{\sin(\theta)\sin(\omega\Delta t)}{\omega} \\ \frac{v_f\sin(\theta)}{\omega}\left(\Delta t\cos(\omega\Delta t) - \frac{\sin(\omega\Delta t)}{\omega}\right) + \frac{v_f\cos(\theta)}{\omega}\left(\Delta t\sin(\omega\Delta t) + \frac{\cos(\omega\Delta t)}{\omega} - \frac{1.0}{\omega}\right) \\ v_f\sin(\theta)\cos(\omega\Delta t) + v_f\sin(\omega\Delta t)\cos(\theta) \\ 0 \\ t \\ \omega \end{bmatrix}. \end{aligned} \tag{3.12}$$

$$\mathbf{A} = \frac{\partial f(\mathbf{x}_k, \mathbf{u}_k)}{\partial \mathbf{x}} = \begin{bmatrix} 1 & 0 & \cos(\omega\Delta t)\left(\frac{v_f}{\omega} - \frac{v_f}{\omega}\right)\cos(\theta_{k-1}) - \frac{v_f}{\omega}\sin(\omega\Delta t)\sin(\theta_{k-1}) \\ 0 & 1 & \cos(\omega\Delta t)\left(\frac{v_f}{\omega} - \frac{v_f}{\omega}\right)\sin(\theta_{k-1}) + \frac{v_f}{\omega}\sin(\omega\Delta t)\cos(\theta_{k-1}) \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

3.3 PREREQUISITES FOR LEARNING

In order to learn the dynamics or kinematics of a vehicle the structure of the equations needs to be taken into consideration. Most models define the next state of a vehicle in terms of the previous or base state. This ensures that as the vehicle's motion progresses the variables are updated to reflect the most recent state. As such, learning a vehicle's motion model can be considered a regression problem where the predicted vehicle state is dependent on the previous state.

Learning how states transition from one to the next requires that each state variable be used during training along with the vehicle's kinematics, higher-order dynamics and control variables. However, the variables are difficult to calculate analytically for large training durations. Furthermore, any sensor used to capture the information will only be relative to the sensor's placement on the vehicle, thus offering subjective information. Since higher-order variables such as velocity, acceleration, torque and inertia can be related to the base variables, one can eliminate the higher-order variables from consideration during the learning process. Thus removing the higher-order variables during learning required that the dynamics be learned implicitly.

Most of the higher-order variables depend on the change in time and can therefore be exploited during the learning process. By using multiple previous states of the lower-order variables, a reasonable representation of the higher-order variables should be learnable. As an example, consider a vehicle's acceleration, which is dependent on the change in velocity over time ($\frac{dv}{dt}$) and therefore dependent on the change in position ($\frac{d^2x}{dt^2}$). If multiple vehicle positions are known at certain time instances, the velocity and acceleration can be calculated. Similarly, if multiple previous states are included during training, one should be able to learn the interaction between the variables. Thus, the learning algorithm must be able to satisfy (3.14), where $\mathbf{x}_k$ is the state of the vehicle, the function $f(\mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n})$ is some abstract function that was trained to represent a vehicle's motion and n is the number of previous states included.

$$\mathbf{x}_k = f(\mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}). \tag{3.14}$$

3.3.1 Relaxation of the constraints

The previous section provided a methodology to learn a vehicle's motion model using only the base-state variables. However, as noted with the differential drive's kinematics, the motion depends on the type of control provided as well as the vehicle's structure. A logical alternative to the previous approach would therefore be to include control variables as input during training.

Note that any control input will affect the base-state variables (or pose) and will therefore also be contained implicitly within the trajectory of a vehicle, as explained in the previous section. However, the learning process may be overburdened in learning the exact relationship between all the base state variables. In such a case, including control variables may reduce the number of relationships that need to be learned by the algorithms to provide accurate predictions.

Equation 3.15 therefore provides an alternative function that can be used to learn a vehicle's motion. From the function one can see that only the state's control input ($\mathbf{u}_k$) is added. As an example, the control can be provided as the previous state's forward and rotational velocity to match a differential drive's kinematic model.

$$\mathbf{x}_k = f(\mathbf{u}_{k-1}, \mathbf{x}_{k-1}, \mathbf{x}_{k-2} \dots, \mathbf{x}_{k-n}). \tag{3.15}$$

3.4 MODEL LEARNING USING NEURAL NETWORKS

3.4.1 Primary approach

Learning an abstract function such as (3.14) can be implemented with NNs. By using a TDL-NN configuration, a number of previous states can be kept in memory to ensure that the higher-order variables are modelled during estimation [66]. TDL-NNs are very similar to recurrent neural nets (RNN) due to the fact that both keep some history of the previous inputs. The difference between the two is that RNNs keep some function of the previous inputs as memory, while TDLs keep the actual inputs as memory [84, 20].

In general, a TDL-NN can be described using (3.16) where $\mathbf{x_k}$ is the state vector, $\mathbf{N_h}$ and $\mathbf{N_o}$ are the hidden and output weight matrices and $\tanh(\cdot)$ and $\mathbf{s}$ are the activation functions and scale factors, respectively. The variables encapsulated by the state will therefore only use the base variables. For 2D motion only the vehicle's pose will be considered as the state (i.e. $[x, y, \theta]$).

$$\begin{aligned} \mathbf{x}_k^{NN} &= FFNN\left(\mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}, \mathbf{N_h}, \mathbf{N_o}\right) \\ &= \tanh\left(\tanh\left(\frac{\mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}}{\mathbf{s}}\mathbf{N_h}\right)\mathbf{N_o}\right)\mathbf{s}. \end{aligned} \tag{3.16}$$

Figure 3.3 provides the general configuration for the TDL-NNs where the estimate is dependent on the three previous states. As shown in the figure, the memory of the NN is assumed to be internal to the NNs structure by making use of a shift register. Consequently, the NN operates in much the same way as an Elman net [84], the difference being that instead of using the outputs from the hidden layer as memory, the actual states are used as memory.

3.4.2 Alternative approach

An alternative network training approach is also possible using (3.15). A similar reasoning as the primary approach is followed, with control added as input to the NN. For completeness (3.17) provides the function used by the NNs and Figure 3.4 the corresponding network structure. Note that the control input is not included in the shift register.

$$\begin{aligned} \mathbf{x}_k^{NN} &= FFNN\left(\mathbf{u}_{k-1}, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}, \mathbf{N_h}, \mathbf{N_o}\right) \\ &= \tanh\left(\tanh\left(\frac{\mathbf{u}_{k-1}, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}}{\mathbf{s}}\mathbf{N_h}\right)\mathbf{N_o}\right)\mathbf{s}. \end{aligned} \tag{3.17}$$

3.4.3 Training

The advantage of using a TDL is that one avoids the vanishing gradient problem [90] that occurs in other RNNs during training. Also, as the structure remains feed-forward, any standard training algorithm can be applied to learn the model. A third advantage of the structure is that all the input states can be calculated before training commences, removing the need to calculate the contents of

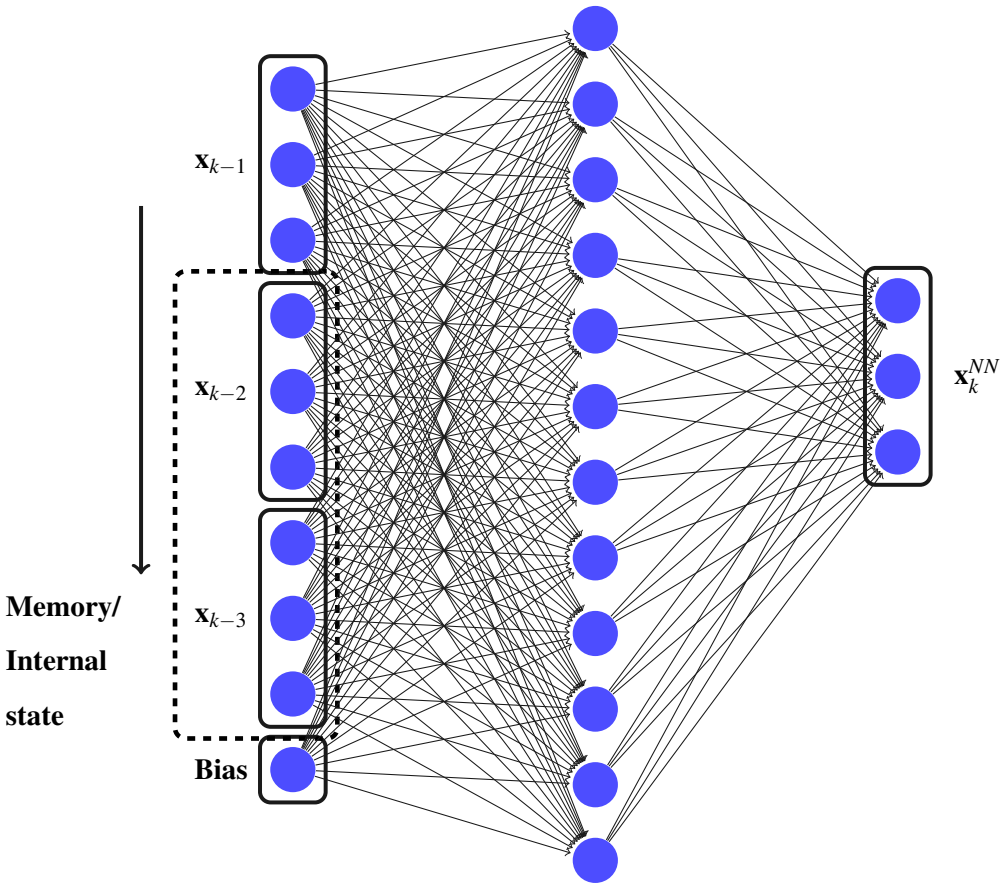


Figure 3.3. Basic TDL-NN configuration where the output is dependent on three previous states.

the shift-register during training. Thus either batch or online training methods can be used while still maintaining consistency.

In order to train the NNs, a gradient-descent algorithm (back-propagation) was implemented [91]. As is conventional with the back-propagation algorithm, both learning-rate and momentum factors were included in the training algorithm to limit overshoot and oscillations. Algorithm 3.1 provides the general procedure for an online back-propagation approach using a tanh activation function where:

- $\mathbf{A}_h, \mathbf{A}_o$ are the hidden and output layer's activations,
- $\mathbf{W_k}$ and $\mathbf{W_j}$ are the quantities by which the weight matrices have been adjusted,
- $\mathbf{G}_o$ and $\mathbf{G}_{res}$ are the output and residual gradients,

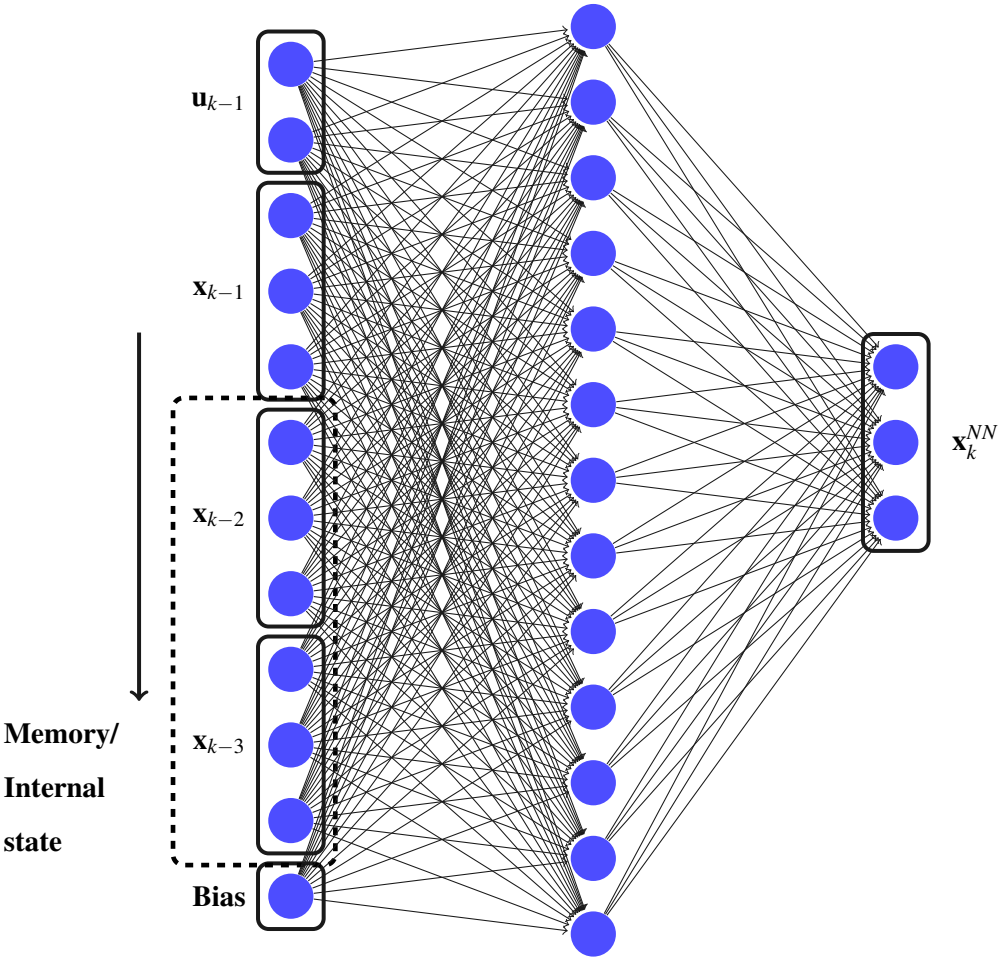


Figure 3.4. Basic TDL-NN configuration where the output is dependent on the control and three previous states.

- l_r is the learning rate,
- m is the momentum,
- n_s is the number of training samples.

To increase robustness of the training algorithm further, an annealing factor was included that decreased both the learning-rate and momentum factors as the training epochs progressed [92]. The reason for including the annealing factor is that the NNs will eventually reach and oscillate around a local minimum. By lowering the amount of adjustment to the weights, the NNs should be able to reach a new local minimum, bringing NNs closer to an optimal solution. Thus systematically decreasing the adjustment factors will lead to a lower convergence value and consequently better representations for

Algorithm 3.1 Back-propagation

1: **for** i in n_s **do**
2: $\mathbf{A}_h, \mathbf{A}_o = FFNN(input[i], \mathbf{N_h}, \mathbf{N_o})$
3: error $= target - \mathbf{A}_o$
4: SquaredError $+= error^2$
5: $\mathbf{G}_o = \frac{\partial x}{\partial t}(\tanh(\mathbf{A}_o))$
6: $\delta_k = \mathbf{G}_o \times error$
7: $\mathbf{E}_k = \mathbf{A}_h \times \delta_k$
8: $\mathbf{W_k} = (-1) \times l_r \times \mathbf{E}_k + m \times \mathbf{W_k}$
9: $\mathbf{N_o} += \mathbf{W_k}$
10: $\mathbf{G}_{res} = \delta_k \times \tanh(\mathbf{A}_h)$
11: $\delta_j = \mathbf{G}_{res} \times (\mathbf{N_o} \times \delta_k)$
12: $\mathbf{E}_j = input[i] \times \delta_j$
13: $\mathbf{W_j} = (-1) \times l_r \times \mathbf{E}_j + m \times \mathbf{W_j}$
14: $\mathbf{N_h} += \mathbf{W_j}$
15: **end for**

the actual model.

One drawback, however, is that the annealing factors could adjust the learning-rate and momentum factors too quickly, leading to slower convergence and (possibly) worse models. To limit the adverse affects that the annealing factor may have on training, the annealing factor was only applied at intervals during training. As a result, the NN should reach a relatively stable error before these factors are lowered. Equation 3.18 provides the equations used to anneal each of the factors systematically. Let

$$l_r(curr) = \frac{l_r(prev)}{1 + \frac{k}{T}}, \tag{3.18}$$

where:

- $l_r(curr)$ is the current learning rate,
- $l_r(prev)$ is the original learning rate,
- k is the current epoch number,
- T is the annealing factor (epochs).

To confirm that the general back-propagation algorithm works as expected, unit tests were implemented that solved a number of simple learning problems. These test problems included the XOR-problem, learning cosines and an unbounded linear problem. The functions used to verify that the back-propagation algorithm was able to find a solution were

$$y_1 = x_a \oplus x_b, \tag{3.19}$$

$$y_2 = \sin(\omega t), \tag{3.20}$$

$$y_3 = \alpha t - \beta. \tag{3.21}$$

where:

- x_a and x_b is the input to NN (binary symbols of either $+1$ or -1),
- t is the input to the NN (floating point values),
- *omega* is the rotational velocity (chosen as π),
- α and β where arbitrary constants (chosen as 1.8 and 2.4),
- y_1 to y_3 is the output of the respective NNs.

3.4.4 Activation functions

The type of activation function used during training influences the complexity of functions that can be learned. For gradient-based learning approaches the most common activation function applied is the tanh function. Other functions that are also often used are the sigmoid, linear, sinusoid and rectified linear unit (ReLU) activation functions [93]. Each of the functions are used primarily based on the type of information available as well the type of outputs required.

The sigmoidal activation functions are commonly applied when input data is between the interval $[0,1]$, while the sinusoid activation functions are commonly used when the input values form a repeating pattern. Linear and ReLU activation functions are adopted when the input data range is unknown or when the data cannot be properly scaled. The difference between the linear and ReLU activation function is that the linear activation's working interval is $[-\infty,\infty]$, while the ReLU function's interval is $[0,\infty]$

While the aforementioned activation functions significantly alter the behaviour of the NNs, it should be noted that only the sigmoidal activation function offers similar performance with regard to the tanh function using gradient descent algorithms. However, as the NNs needed to solve a regression problem, other functions could potentially be more appropriate. Evaluating the performance of different activation functions in a model learning context was therefore considered prudent to ensure correct operation. Specifically, the performance of a linear activation function was compared to the tanh activation during training. For completeness, the activation functions were defined as

$$\Phi(x) = \tanh(x), \tag{3.22}$$

$$\Phi(x) = 0.1x, \tag{3.23}$$

where:

- x is the input.
- $\Phi(x)$ is the activation's output.

Note that the linear activation included a small scaling factor to ensure that large training errors were underestimated. An added benefit was that the scaling factor limited the corresponding gradient error of the NN's weight adjustment, increasing the likelihood of convergence.

3.5 SLAM WITH THE LEARNED MODEL

The previous section described the general procedure employed to train abstract motion models. One important consequence of the training procedure is the fact that the previous states need to be known in order to provide predictions. Many practical applications, however, cannot provide direct estimates for the state with reasonable accuracy. Consequently, most localisation algorithms fuse the sensor information using some predefined measurement model that relates the measurements to the actual state.

Sensors usually capture motion as either raw odometry, velocity, inertial measurements or by comparing visual landmarks. Recursive Bayes filters and HMMs [94] are some of the most widely accepted statistical methods to infer the actual unobserved state through measurements. Specifically, the EKF

[28] has been widely adopted in many systems to provide estimates for the state. The subsequent sections provide the implementation details concerning an EKF-SLAM algorithm that makes use of the learned model.

3.5.1 Derivation of the transitional model

The EKF algorithm is based on a first-order HMM that assumes that the current state is only dependent on the previous state, while the measurement model is only dependent on the current state. Furthermore, the first-order HMMs assume that the probability distribution of the system is Gaussian. The learning strategy described previously, however, is dependent on more than one previous state and can be considered an n^{th}-order Markov model. To use the learned model in an EKF-SLAM implementation, the N^{th}-order Markov model first needs to be related to a first-order Markov model.

Two possible models are defined to learn vehicle motion models, each dependent on the vehicle's state. However, the second approach has additional control input that needs to be taken into account. The following therefore provides the reasoning used to incorporate both models into an EKF-SLAM implementation followed by any assumptions required to produce a tractable solution.

3.5.1.1 Primary approach

For an N^{th}-order Markov model joint probability of any number of states is defined as

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1), \ldots, p(\mathbf{x}_{n-1}|\mathbf{x}_{n-2}, \ldots, \mathbf{x}_1) \prod_{m=n}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}, \ldots, \mathbf{x}_{m-n}), \tag{3.24}$$

where $p(\mathbf{x}_m|\mathbf{x}_{m-1}, \ldots, \mathbf{x}_{m-n})$ is the function to be learned. The complexity of recursively providing estimates for the joint probabilities therefore scales with the number of previous states used. However, as noted in Section 3.3, the previous states are mainly used to provide information to model the higher-order dynamics in the system. By using the memory to model the higher-order dynamics along with the assumption that the actual model remains unknown, we assume that the NN's prediction,

$$\mathbf{x}_m^{NN} = FFNN(\mathbf{x}_{m-1}, \mathbf{r}_m, \mathbf{N_h}, \mathbf{N_o}), \tag{3.25}$$

is dependent on the previous state and some internal state ($\mathbf{r}_m$). Thus the joint probability can be redefined using (3.25), where the internal state represents the information used to model the higher-order dynamics as

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M | \mathbf{r}_1, \ldots, \mathbf{r}_M) = p(\mathbf{x}_1|\mathbf{r}_1) p(\mathbf{x}_2|\mathbf{x}_1, \mathbf{r}_2), \ldots, p(\mathbf{x}_{n-1}|\mathbf{x}_{n-2}, \mathbf{r}_{n-1}) \prod_{m=n}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}, \mathbf{r}_m). \tag{3.26}$$

The internal state therefore encapsulates the external forces that act on the system without explicitly defining the variables. In essence, the NN becomes a "black box" estimator from the EKF's viewpoint. Furthermore, because the previous vehicle states are defined as internal to the NN, conditional independence can be assumed between the NN's internal state and the current state's prediction. Hence the following simplification can be made for the EKF:

$$p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{r}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}). \tag{3.27}$$

Using the aforementioned assumption with (3.26), the joint probability of the state can be simplified to a first-order Markov model:

$$\begin{aligned} p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M | \mathbf{r}_1, \ldots, \mathbf{r}_M) &= p(\mathbf{x}_1) p(\mathbf{x}_2|\mathbf{x}_1), \ldots, p(\mathbf{x}_{n-1}|\mathbf{x}_{n-2}) \prod_{m=n}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}) \\ &= p(\mathbf{x}_1) \prod_{m=2}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}). \end{aligned} \tag{3.28}$$

Note that a trade-off now occurs whereby the joint probabilities become tractable at the cost of knowledge of the state's transition (see Section 3.5.1.3).

3.5.1.2 Alternative approach

Adding control to the NN still results in an n^{th}-order Markov model. The difference is that both the state and control variables needs to be included in the model. The full joint probability for all the states and control can therefore be defined as

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M, \mathbf{u}_0, \mathbf{u}_2, \ldots, \mathbf{u}_{M-1}) = p(\mathbf{x}_1|\mathbf{u}_0)p(\mathbf{x}_2|\mathbf{x}_1, \mathbf{u}_1, \mathbf{u}_0), p(\mathbf{x}_3|\mathbf{x}_2, \mathbf{x}_1, \mathbf{u}_2, \mathbf{u}_1, \mathbf{u}_0)\ldots,$$
$$p(\mathbf{x}_{n-1}|\mathbf{x}_{n-2}, \ldots, \mathbf{x}_1, \mathbf{u}_{n-2}, \ldots, \mathbf{u}_0)\times$$
$$\prod_{m=n}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}, \ldots, \mathbf{x}_{m-n}, \mathbf{u}_{m-1}, \ldots \mathbf{u}_0). \tag{3.29}$$

However, from (3.17) only the current control is used. Furthermore, the past controls are independent of the current control conditioned on the current and previous state. Hence the current control is only dependent on the current and previous state and therefore conditionally independent of past controls, leading to the following simplification:

$$p(\mathbf{x}_m|\mathbf{x}_{m-1}, \ldots, \mathbf{x}_{m-n}, \mathbf{u}_{m-1}, \ldots \mathbf{u}_0) = p(\mathbf{x}_m|\mathbf{x}_{m-1}, \ldots \mathbf{x}_{m-n}, \mathbf{u}_{m-1}). \tag{3.30}$$

Substituting (3.30) into (3.29) the full joint probability becomes:

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M, \mathbf{u}_0, \mathbf{u}_2, \ldots, \mathbf{u}_{M-1}) = p(\mathbf{x}_1|\mathbf{u}_0)p(\mathbf{x}_2|\mathbf{x}_1, \mathbf{u}_1), p(\mathbf{x}_3|\mathbf{x}_2, \mathbf{x}_1, \mathbf{u}_2)\ldots,$$
$$p(\mathbf{x}_{n-1}|\mathbf{x}_{n-2}, \ldots, \mathbf{x}_1, \mathbf{u}_{n-2}) \times \prod_{m=n}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}, \ldots, \mathbf{x}_{m-n}, \mathbf{u}_{m-1}). \tag{3.31}$$

Defining the NN to contain the memory as an internal state with an added dependency on control results in

$$\mathbf{x}_m^{NN} = FFNN(\mathbf{x}_{m-1}, \mathbf{u}_{m-1}, \mathbf{r}_m, \mathbf{N_h}, \mathbf{N_o}). \tag{3.32}$$

Assuming again that the internal state is conditionally independent, the current vehicle state can be simplified to

$$p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{u}_{n-1}, \mathbf{r}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{u}_{n-1}). \tag{3.33}$$

Substituting (3.33) into (3.31) then produces the following

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M, \mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_M | \mathbf{r}_1, \ldots, \mathbf{r}_M) = p(\mathbf{x}_1|\mathbf{u}_1) p(\mathbf{x}_2|\mathbf{x}_1, \mathbf{u}_2), p(\mathbf{x}_3|\mathbf{x}_2, \mathbf{u}_3) \ldots, \\ p(\mathbf{x}_{n-1}|\mathbf{x}_{n-2}, \mathbf{u}_{n-1}) \times \prod_{m=n}^{M} p(\mathbf{x}_m|\mathbf{x}_{m-1}, \mathbf{u}_m). \quad (3.34)$$

The resulting model (3.34) is the same as the previous result, with an added dependency in the form of the current control. Both approaches can therefore be integrated into a standard EKF algorithm using aforementioned assumptions.

3.5.1.3 Consolidating the NN and EKF

The previous results do contain one caveat that still needs to be taken into account. By assuming that the learned model contains some unknown internal state, the NN is reduced to a black-box estimator. This, however, is not problematic, as previous work by [21, 81, 82, 85, 88] demonstrated that a NN can operate in a recursive Bayesian filter. To make the solution tractable, the transitional model is simply assumed to be a linear function of the TDL-NN as

$$\begin{aligned} \hat{\mathbf{x}}_k &= f(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}) \\ &= \mathbf{x}_k^{NN} \\ &= \begin{bmatrix} x_k^{NN} \\ y_k^{NN} \\ \theta_k^{NN} \end{bmatrix}, \end{aligned} \quad (3.35)$$

where

- $\mathbf{u}_k$ represents the control,
- $\hat{\mathbf{x}}_k$ the EKF prediction,
- $\mathbf{x}_k^{NN}$ the NN's prediction,
- x_k^{NN}, y_k^{NN} and θ_k^{NN} the 2D pose predicted by the TDL-NN.

One important characteristic to note is that the EKF does not provide the actual predictions. Rather the NN provides the prediction while the EKF simply incorporates the predictions into the transitional model. As such the EKF has no knowledge of the NN's model, leading to the assumption of a linear

transitional model. The Jacobians of the transitional model (3.35) are therefore also linear

$$\mathbf{A} = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.36}$$

$$\mathbf{U} = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{u}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.37}$$

having no effect on the other state variables. The implication is that the EKF is only used to fuse the sensor information from the measurement model to recursively update the state. Using these requirements, one can now construct a first-order HMM for the transitional and measurement models. This, in turn, allows the EKF to provide updates to the predicted state.

3.5.2 Measurement model definition

In order to correct the predictions provided by the NN, a measurement model needs to be defined. Any measurement model that can be related to the state can be used. However, the purpose of the learned model is to provide improved predictions during localisation and mapping. Hence using 3D landmarks for the measurement model was selected, as landmarks are most representative of a complete SLAM system.

Landmarks in a SLAM context refer to a structure in 3D space that can be uniquely recognised. These structures can be tracked while a vehicle moves through an environment to provide estimates of the vehicle's absolute pose within a global frame. In addition, landmarks can be used to detect loop-closures in an environment when a scene is revisited. Usually the absolute global position of landmarks cannot be measured directly. However, the measurements are observed as some relative distance and angle from the sensors. The following therefore defines how the landmarks are measured:

$$\mathbf{y}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \tag{3.38}$$

$$\bar{\mathbf{y}}_i = \mathbf{y}_i \ominus (\mathbf{x}_v \oplus \mathbf{x}_s). \tag{3.39}$$

where

- $\mathbf{y}_i$ is the global coordinates for the landmark,
- $\bar{\mathbf{y}}_i$ is the landmark's coordinate relative to the vehicle's pose,
- $\mathbf{x}_v$ is the vehicle's pose,
- $\mathbf{x}_s$ is the transform from the sensor to the vehicle's pose,
- $\oplus$ and $\ominus$ are the motion composition operators (see Addendum B).

To make use of the relative landmarks a standard range-bearing measurement model (range, yaw and pitch) can be employed to update the state, given by

$$\begin{aligned} \mathbf{z}_i &= h(\bar{\mathbf{y}}_i, \mathbf{v}_k) \\ &= \begin{bmatrix} \sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^2} \\ \arctan(\frac{\bar{\mathbf{y}}_i}{\bar{\mathbf{x}}_i}) \\ -\arctan(\frac{\bar{\mathbf{z}}_i}{\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2}}) \end{bmatrix} + \begin{bmatrix} v_r \\ v_\theta \\ v_\phi \end{bmatrix}. \end{aligned} \tag{3.40}$$

Thus the measurement model depends on the relative landmark measurement, the pose, as well as some noise ($\mathbf{v}_k$). Using the equations for the relative landmark's range-bearing measurements, the Jacobians can be calculated as

$$\mathbf{H} = \frac{\partial h(\bar{\mathbf{y}}_i, \mathbf{v}_k)}{\partial \bar{\mathbf{y}}_i}$$
$$= \begin{bmatrix} \frac{\bar{\mathbf{x}}_i}{\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^2}} & \frac{\bar{\mathbf{y}}_i}{\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^2}} & \frac{\bar{\mathbf{z}}_i}{\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^2}} \\ -\frac{\bar{\mathbf{y}}_i}{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2} & \frac{\bar{\mathbf{x}}_i}{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2} & 0 \\ \frac{\bar{\mathbf{x}}_i \bar{\mathbf{z}}_i^2}{\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2}\left(\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^4\right)} & \frac{\bar{\mathbf{y}}_i \bar{\mathbf{z}}_i^2}{\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2}\left(\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^4\right)} & -\frac{2\bar{\mathbf{z}}_i\sqrt{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2}}{\bar{\mathbf{x}}_i^2 + \bar{\mathbf{y}}_i^2 + \bar{\mathbf{z}}_i^4} \end{bmatrix} \tag{3.41}$$

$$\mathbf{V} = \frac{\partial h(\bar{\mathbf{y}}_i, \mathbf{v}_k)}{\partial \mathbf{v}}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.42}$$

However, one should note that the actual state is encapsulated within the relative landmarks. Thus each relative landmark position should be replaced with the actual state and landmark variables and the partial Jacobians calculated. Detecting and associating the landmarks in the map is a significant problem that still needs to be addressed. However, as data association and landmark handling was not the primary focus of the research, details regarding the implementation is not provided in this chapter. Instead, the reader is referred to Addendum A for further information.

3.5.3 EKF-SLAM implementation

The following section provides a general overview of the EKF-SLAM implementation. Note that the implementation follows the general guidelines specified in [95]. EKF-SLAM is based on the assumption that a large state variable describes both the landmarks and actual vehicle motion, thus forming a complete representation of the map as well as the vehicle state. In general the state for an EKF-SLAM implementation is defined as:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x_v} \\ \mathbf{y_1} \\ \vdots \\ \mathbf{y_L} \\ \end{bmatrix}, \tag{3.43}$$

where $\mathbf{y_i}$ is the i^{th} landmark and $\mathbf{x_v}$ is the vehicle pose. An important characteristic for the EKF-SLAM's transitional model is the static landmark assumption, which simplifies calculation of the Jacobian matrices. As such the Jacobian is defined as:

$$\frac{\partial f}{\partial \mathbf{x}}|_{(3L+3)\times(3L+3)} = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{x_v}}|_{3\times3} & 0|_{3\times3} & \dots & 0|_{3\times3} \\ 0|_{3\times3} & I|_{3\times3} & \dots & 0|_{3\times3} \\ \vdots & \vdots & \ddots & 0|_{3\times3} \\ 0|_{3\times3} & 0|_{3\times3} & \dots & I|_{3\times3} \end{bmatrix}, \tag{3.44}$$

where the subscript $|_{3\times3}$ indicates the size of the matrix. The covariance associated with the state can be defined using (3.45), where sub-covariance matrices are each individual landmark's covariance with itself, other landmarks or the vehicle pose.

$$\mathbf{P_k}|_{(3L+3)\times(3L+3)} = \begin{bmatrix} \mathbf{P_{xx}}|_{3\times3} & \mathbf{P_{xy1}}|_{3\times3} & \dots & \mathbf{P_{xyL}}|_{3\times3} \\ \mathbf{P_{y1x}}|_{3\times3} & \mathbf{P_{y1y1}}|_{3\times3} & \dots & \mathbf{P_{y1yL}}|_{3\times3} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P_{yLx}}|_{3\times3} & \mathbf{P_{yLy1}}|_{3\times3} & \dots & \mathbf{P_{yLyL}}|_{3\times3} \end{bmatrix}. \tag{3.45}$$

Using the aforementioned definitions for the state, covariance, transition and measurement models, the general EKF algorithm can be defined using the regular prediction-update procedure. The generic equations for the two-step algorithm are defined as:

Prediction:

$$\hat{\mathbf{x}}_{k|k-1} = f(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k), \tag{3.46}$$

$$\mathbf{P_{k|k-1}} = \frac{\partial f}{\partial x}\mathbf{P_{k-1|k-1}}\frac{\partial f}{\partial x}^T + \mathbf{W}w_k\mathbf{W}, \tag{3.47}$$

Update:

$$\hat{\mathbf{y}}_{k|k-1} = z_k - h(\hat{\mathbf{x}}_{k|k-1}, 0), \tag{3.48}$$

$$\mathbf{S}_k = \frac{\partial h}{\partial x}\mathbf{P}_{\mathbf{k|k-1}}\frac{\partial h}{\partial x}^T + \mathbf{V}v_k\mathbf{V}, \tag{3.49}$$

$$\mathbf{K}_k = \mathbf{P}_{\mathbf{k|k-1}}\frac{\partial h}{\partial x}^T \mathbf{S}_k^{-1}, \tag{3.50}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k\hat{\mathbf{y}}_{k|k-1}, \tag{3.51}$$

$$\mathbf{P}_{\mathbf{k|k}} = (1 - \mathbf{K}_k\frac{\partial h}{\partial x})\mathbf{P}_{\mathbf{k|k-1}}. \tag{3.52}$$

The computational complexity of the general EKF algorithm is therefore $\mathcal{O}(N^3)$, where N is the size of the state vector given in (3.43). However, by noting that the Kalman innovation matrix ($\mathbf{S}_k$) is scalar with each measurement, the EKF can be implemented to execute in $\mathcal{O}(N^2)$ instead of $\mathcal{O}(N^3)$. As the $\mathcal{O}(N^2)$ EKF-SLAM only allows for a computational speed-up, the procedure used to implement a $\mathcal{O}(N^2)$ EKF-SLAM system is not described. Instead, the reader is referred to [95] for the exact implementation details of an $\mathcal{O}(N^2)$ EKF-SLAM algorithm.

Map expansion is the last component of the EKF-SLAM implementation that needs to be taken into account. For every newly detected landmark, the covariance matrix's row and columns need to be increased and corresponding covariance $\mathbf{P}|_{3\times3}$ added. In addition, the newly detected landmark's global coordinates need to be calculated using the inverse sensor model. Thus by substituting (3.40) into (3.39) the inverse sensor model can be defined as:

$$\mathbf{y}_i = \mathbf{x}_v \oplus \mathbf{x}_s \oplus \begin{bmatrix} z_r\cos(z_\theta)\cos(z_\phi) \\ z_r\sin(z_\theta)\cos(z_\phi) \\ -z_r\cos(z_\phi) \end{bmatrix}, \tag{3.53}$$

and used to calculate the global coordinate.

3.5.4 Full implementation

Now that the SLAM implementation for the learned model has been derived, the full execution process of the algorithm can be explained. Figure 3.5 provides an overview of the block diagram for the model learning approach. As stated in Section 3.4, the TDL-NN made use of a shift register to encapsulate

the previously observed states. Hence only the current state was be added to the NN predict the next vehicle state. If the alternative approach was used, the current control was also added to the NN before calculating the prediction.

The Jacobians were calculated using the state and noise variables, followed by the EKF's prediction and covariance using (3.46) and (3.47). Subsequently, the predicted state and covariances were used along with the landmarks and measurement noise to update state. As with the prediction step, the measurement Jacobians were calculated before updating the vehicle's state using the procedure described in (3.48) to (3.52). Thus the normal EKF-SLAM algorithm was followed, with an added step for the TDL-NN to calculate a prediction.

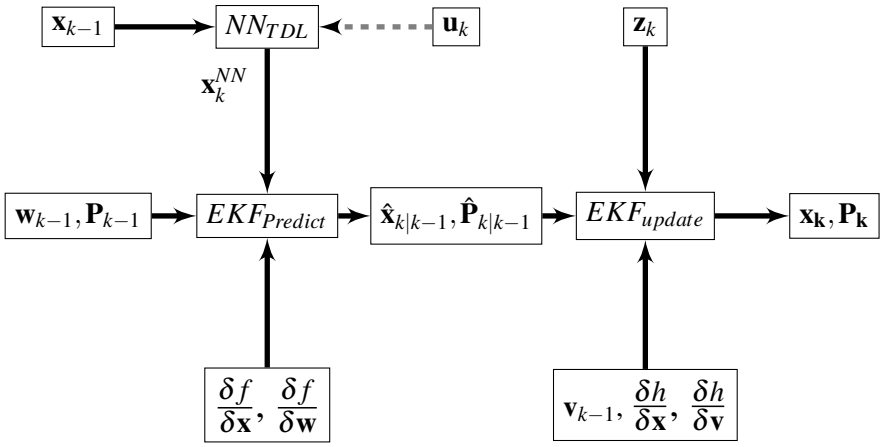


Figure 3.5. General prediction and update procedure once the neural net has been trained. The TDL-NN provides a prediction based on the previous state (and optional control). The prediction is incorporated into the EKF, with its corresponding covariance calculated. Finally, the prediction and covariance are updated using the landmark measurements.

A framework was created within ROS to execute the SLAM implementations. The framework included loading the correct datasets, algorithms and incorporating the NNs within the EKF-SLAM process. Furthermore, both the $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2)$ EKF-SLAM systems were implemented using the *sympy* symbolic math libraries, while the NN implementation made use of the *numpy* libraries. Additional information on the framework can be found in Addendum D.1.

3.6 SUMMARY

A detailed discussion for learning data-derived non-analytical motion models and using the learned models in an EKF-SLAM system was provided in this chapter. Investigation of analytical motion models led to suppositions regarding the required information to learn data-derived non-analytical models. Specifically, the approach used low-order state data with memory (and control) as input data to a TDL-NN implementation.

In addition, the chapter analysed the N^{th}-order Markov process of the TDL-NN and proposed a methodology for incorporating TDL-NN into the EKF-SLAM system. Finally, the full SLAM implementation was described to update the NN predictions with landmark observations.

CHAPTER 4 TRAINING DATA AND EVALUATION METRICS

4.1 CHAPTER OVERVIEW

The previous chapter described the approach followed to learn a motion model and incorporate the model into a SLAM system. However, the type, size and scope of training data required to learn the models have yet to be addressed. In addition, performance metrics for the learned models and the models' parameters need to be considered. The following chapter therefore provides an extensive overview of the training data requirements, evaluation metrics and influential parameters.

For the training data requirements, both real-world and simulated trajectories are discussed in Section 4.2 and Section 4.3, respectively. In each section, changes to the data's format such as scale, sampling rates and any discontinuities that may exist within the data are detailed. In addition, the types of motion required to create simulated datasets are discussed, along with how to diversely cover the problem space with control trajectories.

The second part of the chapter describes the evaluation metrics used to quantify the performance of the learning approach and SLAM implementation. Specifically, the evaluation metrics measuring the suitability of NNs are described in Section 4.4, while Section 4.5 details the metrics used to assess the performance of a SLAM system. In addition, Section 4.6 describes the evaluation procedure proposed to determine the types of motion learned by the model. The nature and information supplied by the metrics in the evaluation procedure are also examined.

Lastly, Section 4.7 characterises the parameters that could influence learning. To this end, the parameters that could affect learning and estimation are identified. Permutations between the parameters are detailed as well as the impact that each parameter has during training. Finally, all the parameters are ranked according to those that have most influence on learning and the sequence of evaluation stipulated.

4.2 REAL-WORLD DATASETS

Real-world datasets containing information of various platforms are freely available for evaluation with SLAM implementations. However, most provide different types of information, depending on the sensors used, the vehicle's drive-type and scale of the map. An overview of frequently used SLAM datasets is therefore provided.

Subsequently, an extensive overview of the TUM Freiburg datasets [35] is provided. Specifically, the format, scale, sampling rates and discontinuities were evaluated to provide suitable training data. Solutions to any problems/inconsistencies in the data are described and removed to ensure an effective learning process.

4.2.1 Available datasets

The New College dataset [96] was gathered in Oxford by a robot driving around the campus and parks for several kilometers. The platform was based on a modified Segway [97] fitted with various sensors. The data available from the platform included a five degree-of-freedom (DOF) trajectory using dead-reckoning, two laser range finders mounted vertically (to measure height ranges), a stereoscopic camera and a five-view omnidirectional camera. A GPS and an IMU sensor were also mounted on the system to provide further information on the vehicle's odometry and position.

A significant difficulty with the New College dataset is that the laser range data cannot be correlated to each other because the ranges were measured in the vertical axis. As such, the stereoscopic camera needed to create a depth map of the environment while the omni-directional camera handled data-association. In addition, the vehicle's odometry, while fairly accurate, does produce a large error in the trajectory, leading the dead-reckoned poses completely astray. While the dead-reckoned poses is

not a problem in itself, as the GPS and other sensors can account for location errors, the pose data is unsuitable for training.

An alternative that was considered was a multi-session dataset captured using the Azimut-3 robot [98, 99, 100]. The Azimut-3 robot operates with four omni-directional wheels that provide raw odometry information as the robot moves around an environment. Both a Kinect sensor and laser scanner were mounted to provide colour and depth information to the system. The platform captured five overlapping maps of an office environment over multiple sessions and are available as ROS *Bag-files*. The sessions were used to test whether SLAM algorithms were able to detect loop closures with previously mapped environments [101]. The vehicle odometry for the Azimut-3 datasets were fairly accurate and could possibly be used for training. However, as the platform was based on a non-standard omni-drive system, it seemed prudent to use a platform that had holonomic constraints that were easier to define.

The Malaga datasets [102] are a collection of datasets captured in urban scenarios. The platform was built into a Citroen C4 and mounted with eight sensors to provide information to the entire system. The sensor platform consisted of a stereo camera, five laser scanners, one IMU and one GPS. Two types of laser scanners were employed in the dataset: three Hokuyo UTM-30LX running at 40 Hz and two SICK LMS-200 laser scanners running at 75 Hz. The two SICK lasers were placed to sense objects in the horizontal plane, while two of the Hokuyo lasers were placed to provide vertical scans. The last Hokuyo scanner was placed pointing forwards and slightly downwards in order to provide scans of the road.

The IMU sensors made use of MEMS technology to provide three-axis acceleration, three-axis angular velocity as well as attitude dead-reckoned 3D pose. Lastly, a low-cost GPS sensor was installed to provide approximate positioning at a rate of 1 Hz. However, some occlusion was present in the datasets owing to the buildings, which led to lost GPS measurements at certain times. As with the New College dataset, the accuracy of the vehicle's trajectory could not be verified, as the GPS's accuracy and frame-rate were not high enough. Furthermore, the trajectory of the dataset is approximately 37 km, making the scale very large for training. Consequently, the Malaga datasets were also insufficient to provide accurate training data.

The TUM Freiburg datasets [35] was captured using a Pioneer robot using a differential drive system. As with the Azimut platform, the Poineer robot used a laser scanner and Kinect to provide depth and colour information, while an IMU sensor provided the inertia. Furthermore, the dataset provided the raw odometry information along with the ground-truth poses of the vehicle. The ground-truth was obtained from a motion capture system in order to compare the predicted poses of SLAM algorithms.

Four sessions of the same environment are available for the dataset as either ROS *Bag-files* or as raw image data with log files containing other sensor information. One session of the data could therefore be used as a test set without concern that specific motions were excluded from the training set. The different sessions, along with the ground-truth provided by the motion capture system, made the datasets ideal for training on real-world data.

4.2.2 Precision

The model learning strategy made use of a number of previous states to estimate the next state of the vehicle model. Ensuring that the training data is as close as possible to the actual vehicle's pose is therefore of significant importance. The ground-truth from the motion capture system provides such data by capturing a Pioneer robot's pose at a rate of 300 FPS [35]. The four datasets of interest are the Robot 360, Robot SLAM, Robot SLAM2 and Robot SLAM3 datasets. The datasets are available in two formats, as a ROS *Bag-file* or a compressed archive containing all the sensor and odometry data. Furthermore, the ground-truth poses of the Freiburg datasets are provided as a text-file containing the time-stamps, translations and quaternions using the format:

```
timestamp tx ty tz qx qy qz qw
```

with a resolution of four floating point values. Motion models are usually expressed in Euler angles instead of quaternions, which allows the model define simpler states. As the preliminary TDL-NN methodology aims to learn 2D motion, only the x, y and yaw measurements are required. A conversion was therefore necessary to provide the data in the correct form. An additional preprocessing step was required by the Freiburg datasets to remove any duplicate measurements from the ground-truth. Most of the datasets contained between two and six duplicate timestamps in the raw data, which could lead

to potential errors. While the source of the duplicate timestamps were unknown, it was speculated that a small error in synchronisation between the motion capture and storage system was the cause.

4.2.3 Input-output training pairs

Using previous states to learn the next vehicle state required that the data be divided into two separate sets: the input consisting of data from index 0 to $n-1$ and the output consisting of data from index 1 to n. Furthermore, as the model learning strategy incorporated multiple previous states, the shift-register needed to be filled. As a result, the input training data of a TDL-NN using m previous states for any given trajectory was in the interval $[m, n-1]$. Similarly the target training values'range was in the interval $[m+1, n]$. Hence for the first training sample for a trajectory the shift register contained states $\mathbf{x}_0$ to $\mathbf{x}_m$, while the desired (target) state is $\mathbf{x}_{m+1}$. During the actual training, the shift-register's content and TDL-NN's target values were calculated beforehand to increase data throughput. Figure 4.1 provides a visualisation of the training data, where $\mathbf{x}_m$ refers to a specific state/pose and **T1** refers to the first training sample.

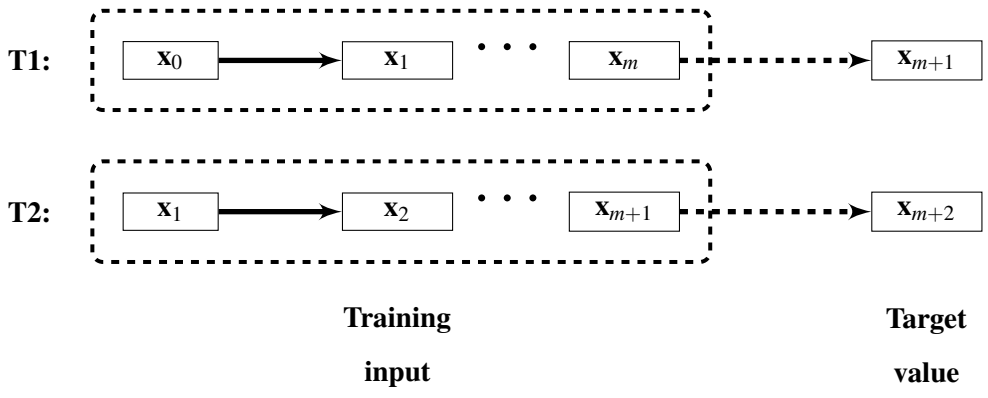


Figure 4.1. Graph of the first two training input/output pairs (**T1** and **T2**) containing m states as memory, where $\mathbf{x}_m$ refers to the m^{th} state/pose. In the figure above, the each training sample represents the contents of the shift-register at a given point during training, while the target value represents the desired output of the TDL-NN.

4.2.4 Scaling

Gradient-based learning in NNs commonly require data to be normalised to the interval $[-1,1]$ for a $\tanh(\cdot)$ activation function. However, since regression is required in order to learn the model, the input data should not be normalised, as normalisation can lead to warping. Specifically, by performing both scaling and shifting on the data, one effectively removes the measurement unit from the data. Even though NNs do not take measurement units into account, the learning problem is dependent on the interactions between different state variables and the variable's measurement unit (e.g. meters and radians). Hence if the input data to the NNs becomes unit-less, then the NNs are unlikely to learn the correct interactions between the state variables.

To limit potentially incorrect interactions being learned by the NNs, only scaling was applied to the input data. By scaling the values the NN would only be able to provide estimates for the vehicle model between the scaling interval. Hence the poses needed to be adjusted to overcome the limited training interval. The simplest method to circumvent the narrow training interval is using local and global poses to handle poses outside of the training data's spectrum. Thus when one of the boundaries is reached, the global location is updated with the current local pose while the local location is reinitialised to zero. Likewise, the shift register within the NN should also be updated with the adjusted previous locations to ensure continuity.

In addition, if the values used to scale the different state variables are too far from each other, warping could potentially ensue. As an example consider that a vehicle can move within a 20×20 meter field during training. Typically the required scaling values would need to be $[20, 20, \pi]$ to perform proper scaling. Hence the scaling values between the position an orientation could have a negative affect during the learning process.

Unfortunately there is no proper way to limit the effects that different scaling values have on the variables. If all the variables are uniformly scaled, one of the state variables may become too small (in the previous example the yaw would be limited to maximum value of 0.1570). Thus the only solution is using training data where the variables have similar intervals. Alternatively, one can use linear activation functions that do not require scaling.

The Freiburg datasets' ground truth data all remain between the interval $[-3.5, 3.5]$ meter for the

position while the orientations remain between $[-\pi, \pi]$ radians. As such, the aforementioned problems should not cause undue warping. Hence, the scaling values were therefore chosen as $[3.5, 3.5, 3.2]$. While the interval is quite small, training data can be provided over most of the activation function's spectrum, which is key for proper training.

4.2.5 Sub-sampling

The ground truth obtained from the motion capture system has a very high frame-rate, as mentioned previously. Commonly odometry measurements are obtained at a frame-rate between 5 and 20 FPS. For the Freiburg datasets the odometry was captured at a frame-rate of 10 FPS. Thus learning to predict at a rate of 300 FPS would prevent the use of vehicle odometry or control during estimation.

Furthermore, owing to the high frame-rate, the difference between consecutive poses become very small. Consequently the NN would estimate the input to be equal to the output. The solution therefore is to select a certain sampling rate and to extract only poses that meet the required time-difference. In this case, the sampling rate was set to 10 FPS in order to match the odometry measurements of the Freiburg datasets. Note that by sub-sampling the data, only 3% of the available data is used. There was therefore a need to increase the number of training examples, as discussed in the next section.

4.2.6 Decimation

As noted in the previous section, sub-sampling the datasets to train on a specific sampling rate has an impact on the amount of training data available. A simple method to increase the amount of training data is to sample at different indexes of each dataset, thereby creating more training examples. Hence, the same trajectory can be sampled at distinct intervals, each having slightly different values than the previous one to provide multiple subsets of the original data. Consequently, all the available data can be utilised. For clarification on the approach, consider the following sequential data:

$$[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2]$$

where the data was captured at a rate of 20 FPS. By sub-sampling the data at a rate of 4 FPS, only the following sequence is available for training:

$[0.0, 0.5, 1.0, 1.5, 2.0]$

which is a fifth of the data available. By sampling at different indexes, the number of training examples can be increased while still maintaining the sub-sampling rate. Thus, assuming sampling is conducted at index $0, 1, 2, 3, 4$, the following subsets can be generated:

0: $[0.0, 0.5, 1.0, 1.5, 2.0]$

1: $[0.1, 0.6, 1.1, 1.6, 2.1]$

2: $[0.2, 0.7, 1.2, 1.7, 2.2]$

3: $[0.3, 0.8, 1.3, 1.8]$

4: $[0.4, 0.9, 1.4, 1.9]$

In each case, the forward sequence of the series is maintained while providing more diverse training data. Note, however, that a maximum of five datasets can be formed in the example sequence, as the data starts to repeat itself. Using the aforementioned approach, more diverse data can be applied for training the NNs without adding repetitive data. Figure 4.2 illustrates how the decimation subsets are formed for each dataset, while Algorithm 4.1 details the procedure used to train the subsets generated from decimation. .

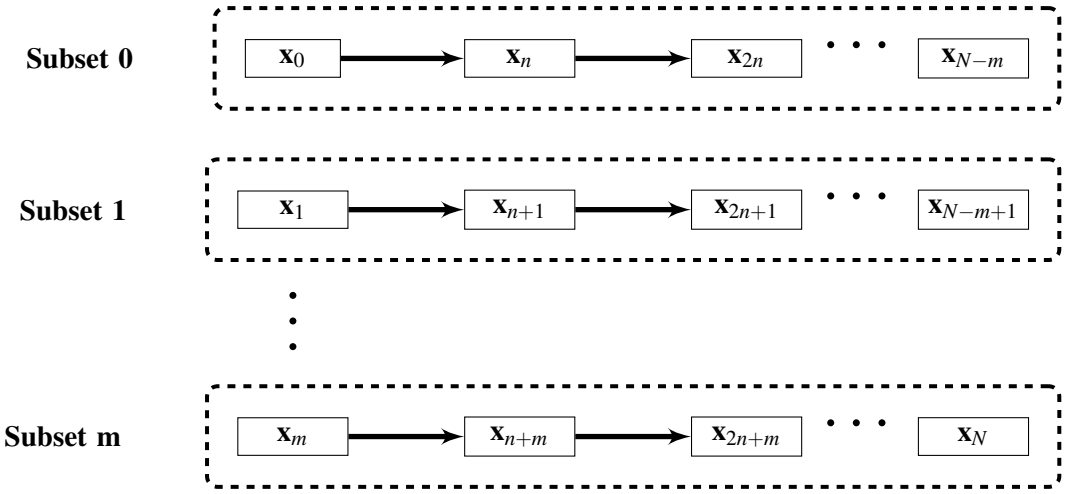


Figure 4.2. Creation of the decimation subsets where $\mathbf{x}_n$ refers to the n^{th} state or pose

Algorithm 4.1 BPSubsetTrain(datasets,maxEpochs, ΔW_i, ΔW_j)

l_r, m = NN.calcAnnealing(l_r, m)
numEpochs = 0
$N_{samples} = \sum_{n=0}^{N} trainDatasets[n].size()$
while currEpochError > minError **AND** numEpochs < maxEpochs **do**
    while len(usedIndexes) < len(datasets) **do**
        usedIndexes, idx = getNewRandomIndex(usedIndexes, len(datasets))
        $sq_{err}, \Delta W_i, \Delta W_j$ = NN.backPropTrain(datasets.in[idx], datasets.out[idx],lr, m, ΔW_i, ΔW_j)
        $t_{err} += \sum(|sq_{err}|)$
    end while
    epochErrors.append($\frac{t_{err}}{N_{samples}}$)
    numEpoch += 1
end while
    return epochErrors, ΔW_i, ΔW_j

4.2.7 Iteratively appending data

The Freiburg robot datasets all contained the same motion dynamics because the same vehicle was used. The only difference between the datasets is that some may contain more information on certain executed motions. A relatively accurate model should therefore be trainable with only one dataset, provided the dataset contains diverse enough motions.. Consequently, additional data from other datasets should only improve training and prediction if information on a certain movement is not encapsulated by the original dataset. Thus, training should be independent of the actual trajectories within the dataset.

In addition, the order that different datasets are added during training should not have an impact on performance. For example, the Robot 360 dataset contains significantly more positive angular velocities than negative velocities, which could cause a bias in the NNs. From the aforementioned conditions, all of the training data need not be included during initial training. Instead, the data can be added incrementally in order to refine the model after every n^{th} epoch using:

$$n = \frac{ep_{max}}{D+\alpha}, \tag{4.1}$$

where:

- ep_{max} is the upper limits set on the epochs,
- D is the number of datasets that was used and,
- α is an addition constant that was set to 2.

The addition constant was arbitrarily selected to ensure that a larger fraction of training would be spent using all the data than any individual increment. A further modification that should be noted with the iterative strategy is that the learning rates could only be adjusted every n^{th} epoch. Thus any additional data is ensured to make smaller adjustments per epoch to the NN's weight and therefore the motion model. Lastly, to ensure that no bias is provided towards a particular dataset, the order in which the datasets are added during training was randomised.

Once all the data had been included, the NNs continued to train until either a minimum error was reached or the maximum number of epochs was reached. For the incremental procedure, early stopping conditions were not included to ensure that all the data was added during training. However, a method that could been used with early stopping was to add the next dataset and continue training when the stopping condition was met. The datasets would then be added until all of the data was included before halting training. Algorithm 4.2 provides the general procedure used to include the datasets incrementally.

The advantage of incrementally adding the training data is that the NNs may initially over-train on the data and create biases, but then refine the weights to account for the new motions as the data becomes available. However, there is a risk that the new data would not be sufficient to escape local minima. Consequently, the effects of iteratively appending the datasets were evaluated in Section 5.2.2.3.

4.2.8 Discontinuities

Pose data contain discontinuities, specifically where the orientation changes between $-\pi$ and π. However, such large orientation changes occur infrequently and would only provide a small number of training samples. Furthermore, NNs using gradient descent algorithms are known to provide sub-

Algorithm 4.2 Train multiple datasets iteratively

```
NN.createFFNN(numInputs, numHidden, numOutputs)
Initialise ΔW_i and ΔW_j
datasetEpochs = maxEpochs / (numDatasets + additionalIterations)
for all i in range(numDatasets + additionalIterations) do
    if i ≤ numDatasets then
        rI = randint(numDatasets)
        trainDatasets.append(datasets[rI])
    end if
    errors, ΔW_i, ΔW_j = BPSubsetTrain(trainDatasets,datasetEpochs, ΔW_i, ΔW_j )
    epochErrors.append(errors)
    vResults, earlyBreak = validateTraining(errors, vResults)
    if earlyBreak is True then
        Break
    end if
end for
```

optimal results for discontinuous data [103]. As a result the discontinuities may have a negative effect during training by biasing the weights towards more extreme changes during estimation.

Splitting the datasets at the discontinuities therefore simplifies the learning process, leaving the Bayesian filter to handle such occurrences. However, by splitting the dataset some of the subsets' size can become very small and lead to unnecessary biasing. To ensure that none of the subsets becomes too small, each of the subset's sizes were compared to the original dataset's size and filtered if the subset contained too few training examples. The following equation was used to evaluate whether any subset was too small:

$$N_{sub} > \frac{N_{orig}}{M_{sub} \times \alpha}, \tag{4.2}$$

where:

- N_{orig} is the number of training examples of the original set,
- N_{sub} is the number of training examples in the current subset,

- M_{sub} is the number of subsets created by the discontinuity split,
- α is some multiplication factor to increase robustness (chosen values were between 2 and 3).

In practice, the NNs could still be exposed to discontinuities and cause significant errors. Three possible solutions exist to handle when discontinuities occur: Allow the EKF to compensate for the errors of the discontinuities, ensure that the NN never observes the discontinuities or re-factor the shift-register once a discontinuity is observed. The first approach requires no modifications, as the assumption is that the landmark observations would correct the predictions until the discontinuity was out of the shift-register. Hence compensation will function similar to a spring-damper system where disturbances induce oscillatory behaviour until the NN stabilises. Consequently, the amount of time that the oscillations occur is directly related to the amount of memory in the shift register.

The second approach can be implemented by assuming that the yaw is not constrained to the interval $[-\pi, \pi]$ by taking the periodic nature of orientation into account. Thus once a phase-change is reached, the values will continue to increase/decrease without adverse effects. Upon completion, the estimated yaw can be re-factored to the desired interval. This method, however, can only be implemented for NNs whose working interval is unconstrained (such as the linear activation functions).

Re-factoring the shift-register once a discontinuity has been detected could be used to avoid the oscillatory behaviour of the first approach. The easiest method to re-factor the shift register is changing all the yaw variables to the currently estimated yaw (after the discontinuity). An important assumption of the approach is that the angular velocity is zero after a discontinuity has occurred. Effectively the NN is partly reinitialised, which could potentially lead to unforeseen behaviour. Furthermore, the approach breaks the consistency of the system by essentially removing the memory from the NN. Partial reinitialisation was therefore eliminated from consideration during testing.

4.3 SIMULATED DATASETS

The real-world datasets provide an abundance of information on a vehicle's motion and encapsulates the dynamics inherent in the vehicle. However, the datasets all contain some form of noise. The most common types of noise present are due to either the sensors' precision or the environment. Another

problem with real-world data is that there may only be a few samples of a particular motion or control. Thus large trajectories with a specific motion/control may not be available in the dataset.

To evaluate the performance of a learning strategy under ideal absolute ground-truth conditions one must therefore employ simulated trajectories. In addition, because the simulated datasets are easily separable, the data can more readily be used to determine whether specific motions and motion types can be learned. The following section details the requirements and use of such simulated datasets in order to determine the types of motion that are learnable under ideal circumstances. Note that the simulated datasets were also subject to sub-sampling, decimation and discontinuity splits, as discussed in the previous section.

4.3.1 Motion model

The first requirement that needs to be addressed is which motion model should be used to create simulated datasets. To ensure that the simulated data are comparable to the real-world datasets, both should have the same form and characteristics of the actual platform. Hence the simulated data needs to be generated based on the Pioneer robot's physical characteristics. The second question to address is whether to use physics engines to create vehicle simulations or analytical models to generate the kinematics or dynamics.

Physics engines are commonly used to simulate the dynamics of vehicles within an environment. Consequently, all of the dynamical properties of both the vehicle and environment needs to be defined in order to create accurate simulations. Gazebo [104] is a physics engine that can communicate with ROS to simulate robots. By default Gazebo uses the open dynamics engine (ODE) [105] to simulate the dynamics, with options to use Simbody [106] and other physic engines.

Gazebo can also incorporate the unified robot description format (URDF) (specified in ROS) to simulate different platforms. In addition, various sensors such as the Kinect, laser scanners and IMUs can be simulated within Gazebo and provide feedback to ROS. However, the standard differential drive controller available in ROS Indigo was found to be insufficient to provide accurate odometry information. Additional information on ROS and Gazebo can be found in Addendum D.2.

Dynamic models make use of a vehicle's weight, dimensions and inertia to take into account both the kinetic and potential energy while the vehicle traverses a trajectory. Hence, a large state is required to model the vehicle's trajectory. Based on the characteristics, a simple dynamic model of the Pioneer robot will provide the most accurate representation of the vehicle's trajectory. However, a dynamic model's control is commonly specified as some input torque on the wheels, allowing the vehicle to undergo acceleration. While there are methods to limit the acceleration by specifying the control as input voltages (a spring and damper system for the motor), the additions generally further complicates the system being modelled.

In comparison, a kinematic model only considers the vehicle's dimensions, with control velocity as input. As a result, a kinematic model is much simpler to generate constant velocity trajectories. Furthermore, the type of control inputs are in the same format as those provided by the Freiburg datasets. While constant velocity trajectories are not completely representative of real-world data, the trajectories do simplify the training data. Furthermore, as the state variables have fewer interactions, the simulated kinematic model's data should be simpler to learn than the dynamic or real-world data

As the aim of the simulated datasets is to simplify the training data to determine the type of motions that could be learned, the kinematic model should be used. It should be noted that once simple constant velocity models (such as the kinematic model) can be learned, simulated data of the dynamic models can be evaluated to establish if more complex models are learnable. One can also extend the approach further by using the learned models on real-world data to determine which model provides a better representation.

4.3.2 Motion types

A vehicle displays different behaviour depending on the type of motion executed. Generating datasets that contain all motion types for a specific vehicle was therefore necessary. For a kinematic model of a differential drive, the following types of motion needed to be representable:

1. Linear or straight-line motion,
2. A curved or circular trajectory,
3. Pure rotational movement and,

4. A stationary vehicle.

From the motion types defined, the curved trajectory was considered the "normal" vehicle motion, with the linear, rotational and stationary motion types considered "special cases". Stationary motion implies that no forces are acting on the vehicle to provide motion and thus should obviously be considered a special case. However, the reason why pure linear or rotational motion were considered special cases may not be as evident.

The first argument why the motions were considered special cases was based on physics, while the second was mathematical. Any object moving in the real world is constrained by external forces acting on the object and will therefore never undergo complete linear or rotational motion. Mathematically, if one investigates the changes that a vehicle's motion undergoes during pure linear and rotational motion, it becomes apparent that not all the state variables are changing (either the orientation or position remains constant). Linear and rotational motion can therefore be considered "simpler" than curved, or arc, motion.

From the motion types defined, simulated datasets were created to provide a comprehensive training set of a vehicle's motion. Combined, the datasets can be used to train a model of all kinematic motion for a specific vehicle. Comparing the simulated data's model to the real-world data's model would then highlight any unforeseen discrepancies. An additional test that can be performed is dividing the simulated datasets into sub-types to train a specific type of motion. The primary advantage of learning a specific motion type is evaluating if the NNs are able to learn each type of motion. Quantifying what the NNs are able to learn individually can therefore be grouped by the motion types. Thus if the NNs are unable to learn a specific type of motion, one can focus on the causes and solutions for that specific type.

4.3.3 Sampling, discontinuities and decimation

For the simulated datasets the trajectory was generated at a rate of 100 FPS with a time interval of 5 seconds. The premise for choosing 5 seconds was to ensure that no trajectory reached full circular motion, thus creating duplicate poses. While a longer time interval could have been selected, a decision was made to limit the amount of data generated during training.

The frame-rate of 100 FPS was selected so that decimation was required to meet a sampling rate of 10 FPS, as in the case of the Freiburg datasets. Since decimation was primarily used to increase the training examples for the real-world datasets, the number of decimation indexes used by the simulated datasets was limited to three (30% of the generated poses). Similarly, the datasets were also split at discontinuities to ease the learning process.

4.3.4 Vehicle control

One question that remains unanswered when generating the simulated datasets is the amount and diversity of data required. Specifically, the extent and range of control velocities required for training need to be addressed. The range of velocities that were used to create training data was set to between 0 and 2.5 m/s for the forward velocities with a rotational velocity between -0.3 and 0.3 rad/s. While not completely matching the Pioneer robot's specifications, the velocities should provide a large enough spectrum of trajectories to reach conclusive results.

The second aspect to address was how extensive the training data within the range should be. Specifically, the number of variations on control required to provide predictions that yield a predefined minimum error needed to be answered. Furthermore, the point at which additional training data no longer added significant improvements needed to be found. Answering these questions required generating datasets that contained varying amounts of control and evaluating the training results. Hence, the number of different controls to train on were chosen as 10, 100, 300 and 418. Addendum F.2 provides the exact control velocities used to generate the training data.

4.3.5 Initial poses

As with the control, the range and extent of the initial poses also needed to be taken into account when generating the simulated data. Ideally the learned model should be able to provide estimates for any control-pose pair provided. However, providing a full dataset for training is practically infeasible. Consequently, a number of limitations were set on the initial poses used for training and testing.

For the simulated data, the initial states were all initialised in the same interval as the Freiburg datasets. Note, however, that a limit on the range of poses was not specified to allow the simulated datasets

to provide estimated trajectories outside the Freiburg dataset's bounds. The range of the trajectory's poses were therefore only limited by the control inputs as well as the time interval.

The number of different initial poses required to train the motion models was another uncertainty that needed to be addressed. Thus datasets were generated with a varying number of initial poses to establish if an optimal number of poses exists and at which point the additional poses stop increasing the NN's accuracy. The number of initial poses generated were selected as 15, 28, 56 and 112. Addendum F.2 provides the exact initial poses used to generate the training data.

4.3.6 Storage

Each simulated trajectory was stored in two text-files, one containing the ground-truth poses and one containing the raw odometry. The ground-truth poses were stored in the same format as the Freiburg dataset's ground-truth data (Section 4.2.2). In comparison, the raw odometry data was used to provide the control and odometry of the vehicle during traversal. Consequently, the raw odometry was only included when control inputs were required by the NNs. The format used to store the data was based on ROS's odometry message:

`timestamp tx ty tz qx qy qz qw vx vy vz` ωx ωy ωz

where `vx` and ωx are the linear and rotational velocity in the x-axis, respectively. The file-name format was generated according to the model used, the pose number and the control number. Two examples are provided below for clarity on the file's format:

- `diffKinematic-p3-c12-groundtruth.txt`,
- `diffKinematic-p12-c02-rawOdom.txt`.

Each file also contained a header describing the format of the file. In each case the header was denoted by a number sign (#) at the start of the line. Directories were also generated because of the large number of simulated datasets. The directories were grouped according to the pose, where each directory contained all the controls applied at a single pose. The directories were named with a "p" followed by the pose number (e.g. `p9`)

4.4 NEURAL NETWORK EVALUATION

To evaluate whether a NN is able to represent a specific function, a number of measures can be employed. The aim is to establish whether the NNs have converged to some minimum error/energy, given the training data supplied. Commonly these measures include training error, the NN's stability and some test vectors to ensure correct operation. A brief description of each evaluation measure is therefore provided in the subsequent sections.

4.4.1 Training errors

Evaluating the error that a NN produces while training is one of the fastest and easiest methods to verify that the NN is converging. The most common error metrics used to determine a NNs performance are:

- Mean Absolute Error (MAE)

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i|, \tag{4.3}$$

- Mean Absolute Percentage Error (MAPE)

$$MAPE = \frac{100}{m}\sum_{i=1}^{m}|\frac{y_i - \hat{y}_i}{y_i}|, \tag{4.4}$$

- Least Squared Error (LSE)

$$LSE = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}|y_i - \hat{y}_i|^2. \tag{4.5}$$

Most of the error metrics are very similar in form, with subtle differences. The MAE measures the average absolute distance between the predicted and actual values, while the MSE measures the deviations of the errors through the squared loss. For an unbiased estimator the MSE therefore measures the variance of the estimator. The LSE is very similar to the MSE, with only the division term included in the metric. The division term is mainly used when calculating the gradient, where the term cancels out the division constant. In comparison, MAPE is used to measure the accuracy as a percentage. There are, however, well-known limitations to MAPE [107].

All of the aforementioned metrics can be graphed over epochs and used to verify whether the NN is converging to a local minimum. Furthermore, by combining the graphs for multiple training iterations

the overall learning ability of the NNs can be gauged. Thus if most of the training data do not reach similar local minima, then there is some problem with the network's configuration. In subsequent experiments the training errors were plotted using a semi-log graph using the LSE metric.

4.4.2 Network stability

An alternative approach to evaluating the NN's convergence is determining stability. To measure stability the actual weights for each neuron were stored and plotted over the number of training epochs. Each neuron's weight can then be plotted as either stability graphs or as delta graphs. The difference between the two is that the stability graphs plot each weight directly over the epochs, while the delta graph plots the difference between the weights over the epochs. Hence as training progresses, each weight in the stability graph should converge around a certain value. In comparison, all the weight changes in the delta graphs should converge at 0 as training progresses. Note that very small deltas do not necessarily imply that the neuron's weight has stabilised. Instead small deltas only indicate that large weight changes over an epoch have been eliminated.

4.4.3 Test vectors

One of the first methods to test whether the NN has learned the model is using test vectors (known input-output pairs). Specifically, one of the datasets not used during training can be used to determine the NN's accuracy by calculating the mean error. An alternative method for evaluating how close the NN's estimates are, is plotting each output variable against the actual output. The plots can then be used to establish if and where the NN's are unable to provide the correct estimates.

The advantage of plotting the output variables against one another is that the graph can easily reveal if the NN is able to learn the model. Furthermore, if the NNs are unable to learn the model, the plot can provide visual information on the type of motion that disrupts convergence during training. The plots can also reveal which of the outputs the NN did not train on properly. If only one or two of the outputs are not estimated correctly, the result could indicate that the input data was insufficient to estimate the transitions properly.

4.5 SLAM EVALUATION

Evaluating SLAM requires that both the trajectory and map's accuracy be measured. Since the aim of the research is to determine if the learned models offer any improvement in localisation, evaluation of the maps were not taken into consideration. Instead, focus was placed on different measures used to establish the accuracy of an estimated trajectory. The first part of this section therefore provides the relevant evaluation metrics for poses and trajectories.

Determining whether the learned model improves the trajectory estimates forms the second part of this section. The metrics stipulates how the models were compared, along with which models were used during the evaluation. Lastly, the data association problem is addressed during SLAM. In particular the use of simulated landmarks is discussed to eliminate association errors.

4.5.1 Trajectory/Pose

The following section describes the most commonly used evaluation metrics for vehicle trajectory and pose.

4.5.1.1 Pose-graphs

A vehicle's pose or trajectory can be evaluated by either creating pose-graphs or through some error metric. Pose graphs provide a visual output of the estimated poses supplied by localisation and mapping. The estimated trajectory for different algorithms can therefore easily be evaluated and compared. Furthermore, an algorithm's performance during specific motions can be interpreted much more readily than when using statistical methods.

Usually only the vehicle's position is supplied by pose-graphs, with a comparison to other algorithms and ground truth. However, in some cases the uncertainty that the algorithm has over the vehicle's position is also included. Commonly, ellipsoids are plotted for each vehicle's pose to demonstrate the amount of uncertainty in the estimates.

In order to test how close the estimated pose is to the actual pose, both the translational and rotational offsets need to be taken into account. The absolute trajectory error (ATE) [108] and relative pose error (RPE) [109, 35] are two methods commonly used to provide analytical comparisons between trajectories. A brief description of each metric is therefore provided.

4.5.1.2 Absolute trajectory error

The ATE metric, as the name suggests, is used to calculate the absolute error between poses by calculating the rigid body transformation between two poses [108]. The rigid body transform between the two trajectories can be calculated by using the centroid of each and solving the least squares problem. Once the transform is calculated, the error is defined as the remainder between the multiplication of the inverse ground truth pose matrix with the rigid body transform and the estimated pose, and given by:

$$\mathbf{F}_i = \mathbf{Q}_i^{-1} \times \mathbf{S} \times \mathbf{P}_i, \tag{4.6}$$

where:

- $\mathbf{F}_i$ is the ATE error at time-step i,
- $\mathbf{Q}_i$ is the ground truth pose at time step i,
- $\mathbf{P}_i$ is the estimated pose at time step i,
- $\mathbf{S}$ is the rigid body transformation that maps the estimated trajectory onto ground truth.

The root mean squared error (RMSE) is regularly used to calculate the translational components of the ATE over a vehicle's trajectory. However, in some instances the mean error is preferred, as the metric is less susceptible to outliers. The RMSE is given by

$$RMSE(\mathbf{F}_{(i:n),\Delta}) = \sqrt{\frac{1}{m}\sum_{i=1}^{n} \|trans(\mathbf{F}_i)\|^2}. \tag{4.7}$$

4.5.1.3 Relative pose error

The RPE measures the accuracy of trajectories during localisation and mapping and is frequently used in visual odometry systems and during loop closures. The advantage of the RPE over the ATE is that the RPE is able to evaluate the trajectories when only sparse, relative pose relations are available as ground truth [109, 35]. Equation 4.8 demonstrates how the RPE can be calculated for a particular time.

$$\mathbf{E}_i := (\mathbf{Q}_i^{-1}\mathbf{Q}_{i+\Delta t})^{-1}(\mathbf{P}_i^{-1}\mathbf{P}_{i+\Delta t}), \tag{4.8}$$

where:

- $\mathbf{Q}_i$ is the ground truth pose matrix at time step i,
- $\mathbf{P}_i$ is the estimated pose matrix at time step i,
- $\mathbf{E}_i$ is the RPE error at time step i,
- Δt is the fixed time interval.

As with the ATE metric, the RMSE over a certain time interval is calculated to provide an overall rotational and translational error, given by

$$TransRMSE(\mathbf{E}_{(i:m)}, \Delta) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\|trans(\mathbf{E}_i)\|^2}, \tag{4.9}$$

$$RotRMSE(\mathbf{E}_{(i:m),\Delta}) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\|rot(\mathbf{E}_i)\|^2}. \tag{4.10}$$

However, the relative errors are only evaluated for a certain time interval. As a result the metric penalises the rotational error more at the beginning of the trajectory than at the end. To provide a robust error that is independent of time, the average error is therefore calculated over all possible time intervals. Note that the computational complexity becomes quadratic because a double summation is required. Thus the RPE errors are commonly sub-sampled before calculating the average RMSE to speed up computation, as shown below:

$$RMSE(\mathbf{E}_{(i:n)}) = \frac{1}{n}\sum_{\Delta=1}^{n} RMSE(\mathbf{E}_{(i:m),\Delta}). \tag{4.11}$$

4.5.2 Model comparison

Establishing whether the learned models improve performance during localisation and mapping requires that the models be evaluated against the analytical models. Both the odometry-based and differential drive models were therefore used as a comparison.

4.5.3 Data association

The data association problem within SLAM can significantly affect the accuracy of SLAM algorithms. Sparse environments and scenes that closely resemble each other can lead to false positive and false negative matches. By using simulated landmarks the data association problem can be bypassed, thus ensuring no false positive or negative measurements are obtained.

Creating simulations that avoid many of the practical problems of data association requires the ground-truth poses, the vehicle's control/odometry as well as the landmark locations. The Freiburg datasets can be used for the simulations, as the datasets contain both ground-truth and odometry control. Vehicle control (in the form of odometry) can therefore be used with the analytical models to provide state predictions as one would implement in a normal SLAM implementation. However, since no landmarks are provided by the dataset, the simulated landmarks were generated. Figure 4.3 provides a plot of the simulated landmarks generated for the simulated SLAM tests.

The measurements for each landmark can be calculated by using the *ground-truth* pose data and the measurement model as described in (3.39) and (3.40). During simulation a simplistic assumption is made regarding landmark measurement: all the landmarks can be detected at each time-instance, regardless of the vehicle's pose. Random noise is induced on the landmark measurements to create a more realistic simulation. In addition, the random noise introduced is uniformly distributed on the covariance interval specified within the parameters.

While simulated landmarks offer valuable information on the performance of each SLAM system, the simulation cannot offer realistic results. Consequently, real-world data association should be included during evaluation. The algorithms used to accomplish real-world data described in Addendum A. In particular, the methods employed to create and match the landmarks using the estimated state of the

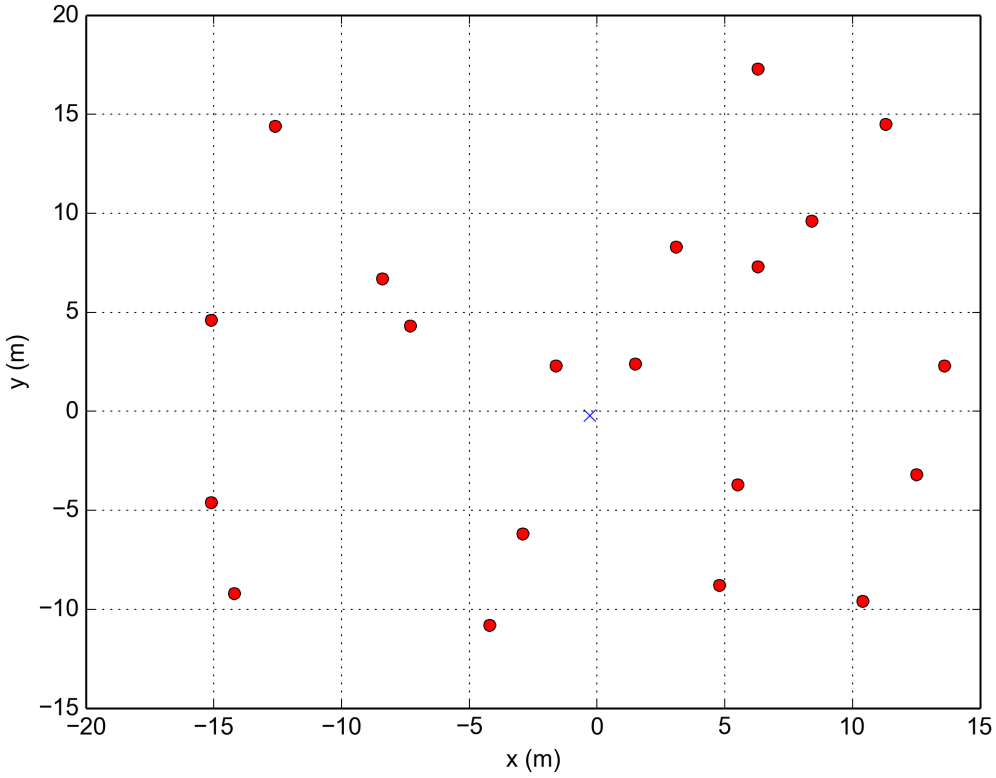


Figure 4.3. The simulated landmark locations.

SLAM system are described. If the learned models can be shown to consistently provide estimates with smaller errors than the analytical models, then the model will have learned a better representation of the vehicle's motion model.

4.6 MODEL LEARNING EVALUATION

Determining whether the motion models can be learned requires the evaluation to encompass a representative sample of the actual motion. Tests therefore need to be conducted over the entire working interval of the model in order to reach statistically relevant conclusions. The following section defines the scope and metrics needed to reach statistically conclusive results.

4.6.1 Requirements and approach

To resolve whether a motion model has been learned requires that the predictions be evaluated. Specifically, the one-forward prediction for many trajectories should be determined with no statistical filter or compensator connected to the output. The approach is similar to the test vectors that NNs commonly make use of, only on a much larger scale and with additional restrictions and requirements.

The scope of the evaluation therefore needs to be representative of the motion model that was learned. A number of factors needs to be taken in account to fully realise the evaluation, most of which are similar to generating the simulated datasets described in Section 4.3. In particular, the learned model should:

- Be able to provide predictions for any type of motion included in the training set,
- Provide accurate predictions for any trajectory whose initial pose starts within the training set's interval,
- Provide accurate predictions for any trajectory generated by control inputs within the training data's scope.

Foremost is ensuring that each type of motion is evaluated for a model. By defining specific motion types within the scope of evaluation, one can determine whether the model is able to provide accurate predictions for each specific type. To establish the error for a specific motion type, errors need to take into account both the initial poses as well as the control used to generate the trajectory.

As stated above, the learned model needs to provide similar errors at any starting pose and control. However, each trajectory tested contains multiple predictions over the entire trajectory. Hence, the overall error for a specific trajectory needed to be determined, and was considered the deepest level of evaluation. Using different pose and control pairs to generate the trajectories therefore provided measurable outcomes for each individual trajectory. However, the trajectory's error cannot quantify the error for the motion types or even a particular control.

The next level used to evaluate the models established the error for a specific control. To provide a measurable outcome for a specific control, the evaluation needed to be independent of any initial pose.

Thus various trajectories needed to be generated where the control remained constant while the initial poses varied. To ensure that the true control's error was approximated, a broad spectrum of initial poses needed to be tested. Combining the trajectories' errors and calculating a summary would then represent the overall error for a specific control. The approach can be extended to multiple controls in order to quantify the errors for different controls.

In addition, the different controls can be combined to form the overall error for the learned model. However, the aim is to determine the types of motion that the models were able to learn. Thus by grouping the tested controls according to motion type and calculating a summary, one can define the error for each motion type. As with the initial poses, a broad enough spectrum of controls needed to be tested to ensure that the true motion type's error is approximated.

The advantage of evaluating the predictions at different levels of depth is that the errors are first defined by a specific trajectory, then controls and finally motion type. As such, both the general performance of the models can be evaluated along with any specific trajectory predictions. More information on the exact poses and controls used to evaluate the motion types can be found in Addendum F.1.

4.6.2 Evaluation metrics

The previous section described the use of many initial poses and controls to generate trajectories that were used to determine if a motion model had been learned. However, owing to the large number of trajectories used during evaluation, the results can easily become biased if a single trajectory/control produces much larger errors, leading to incorrect conclusions. This section therefore investigates the evaluation metrics required to reach conclusive results. In particular, the relevance of statistical components for the errors is established.

Various metrics can be used to define an error between a predicted and target value, some of which were described in Section 4.4. However, each metric only provides information on a certain aspect of the errors. In order to determine whether model learning succeeds, one will need to evaluate various components in order to reach conclusive results.

Furthermore, the learned models can both under- and over-estimate the next state, which can have

dramatic effects on the outcome. Consequently the absolute error was used to define the error for each prediction. The only exception was when creating histograms of a specific trajectory's error. In such a case, knowledge of whether the model over- or under-estimated the movement was more informative. For evaluation at the different levels, the following metrics can be used to measure the various aspects of the model's suitability:

1. MAE (see (4.3))
2. Mean Squared Error (MSE)
$$MSE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i|^2, \tag{4.12}$$
3. Median Absolute Error
$$AE_{median} = median(|y_i - \hat{y}_i|) \tag{4.13}$$
4. Minimum Absolute Error
$$AE_{min} = \min(|y_i - \hat{y}_i|) \tag{4.14}$$
5. Maximum Absolute Error
$$AE_{max} = \max(|y_i - \hat{y}_i|) \tag{4.15}$$
6. Standard Deviation
$$AE_{stdDev} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - MAE)^2} \tag{4.16}$$
7. Variance
$$AE_{var} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - MAE|)^2 \tag{4.17}$$
8. Median Absolute Deviation of the Absolute Error
$$AE_{medianAD} = median(|\hat{y}_i - AE_{median}|) \tag{4.18}$$
9. Skewness
$$AE_{skew} = \frac{3 \times (MAE - AE_{median})}{AE_{stdDev}} \tag{4.19}$$
10. Kurtosis
$$AE_{kurtosis} = \frac{1}{m \times AE_{stdDev}^4}\sum_{i=1}^{m}(\hat{y}_i - MAE)^4 - 3. \tag{4.20}$$

The maximum and minimum absolute error metrics are mainly used to determine if there are any trajectories with errors far larger than those commonly observed. In such cases one can investigate the

individual errors to ascertain where the errors lie and if the error is an outlier. Similarly, the variance and standard deviation can be used to establish the interval of the trajectory's errors. Any cases where the aforementioned metrics vary significantly for different trajectories should therefore be closely investigated.

Using the $AE_{medianAD}$, skewness and kurtosis [110] allow for further statistical analysis that could provide deeper insight into the nature of the learned model. The median absolute deviation (MAD) is a measure of the variability of the absolute errors, comparable to the standard deviation. However, the MAD is more robust to outliers when compared to the standard deviation. As such the $AE_{medianAD}$ can provide a closer estimate of the actual deviations in cases where large outliers are present. The yaw discontinuities and other outliers will therefore not have as large an effect on the $AE_{medianAD}$ metric during the evaluation.

Skewness is defined as the third standardised moment and is used to determine if the weight of a probability distribution is evenly distributed. Thus any large skew values indicates that the errors are not normally distributed. Instead, the mass of the errors will lie either to the right or left of the mean. Similarly, kurtosis is defined as the fourth standardised moment and is used to establish both the peakedness and heaviness of the tails of a distribution. For the tests the excess kurtosis was used, as the metric is normalised to 0 instead of 3.

Large kurtosis values indicate that the distribution has a sharp peak with heavy tails, which is also known as leptokurtic. Under certain conditions, very high kurtosis values could indicate that the errors are the same for the entire trajectory. Conversely, negative kurtosis values (also known as platykurtic) indicate that the distribution is broader with thinner tails. Specifically, a negative kurtosis value of approximately -1 is close to the uniform distribution, while a kurtosis value less than -1.25 is close to a bimodal distribution. In such cases, the kurtosis could indicate that the learned model is unable to provide consistent error predictions for the trajectory.

4.6.3 Implementation

The general procedure used for the model learning metrics are described in Algorithm 4.3 and Algorithm 4.4. In general, Algorithm 4.3 is configured to allow different learned models to be evaluated,

each containing multiple iterations of the same test. Furthermore, multiple motion types can be defined that the system will test. In each case, the algorithm will graph the statistics and calculate a summary of each motion type.

Algorithm 4.3 calcMotionStats(dir, numModels motionTypes, testVel, initPoses, doControl)

```
1:  for currDir in dir do
2:      for all m in motion types do
3:          currVel = testVel.get(m)
4:          for all i in len( numModels) do
5:              allStatistics = testModelKinematics(currDir, initPoses, currVel, doControl, i )
6:              Append all statistics
7:          end for
8:          Process statistics and plot all results on the same graph.
9:      end for
10:     Calculate summary statistics and create bar graphs.
11: end for
```

Algorithm 4.4 is the actual implementation that loads the learned models, generates the kinematic model's test trajectory and calculates the trajectory-level statistics. Each trajectory's individual statistics were calculated and graphed along with the trajectory's histograms.

Lastly, all of the raw estimates were combined and used to calculate the motion type's statistics. The primary reason for combining the raw estimates and calculating the statistics after all the permutations had been tested was to ensure that no bias existed in the data. In particular, metrics that make use of a median (such as the AE_{median} and $AE_{medianAD}$) can contain biased information if each individual trajectory's mean statistics were calculated. Thus, to ensure that the true median over all of the trajectories were calculated, all of the raw estimates were used.

4.7 EXPERIMENTAL PROCEDURES

The internal parameters used during training can significantly affect a NNs performance and learning ability. Similarly the nature of the information encapsulated by the data determines the complexity required to learn the motion model. Hence both the data and the NN's parameters needs to be

Algorithm 4.4 testModelKinematics(currDir, initPoses, currVel, doControl, i)

```
model = loadLearnedModel(currDir, i)
for control in currVel do
    for pose in initPoses do
        kinPoses = vehicleMotionGen.genDiffKin(baselen, pose, control, Δt, numSamples)
        if doControl then
            modelEstimates = testTrajectoryControl(kinPoses, control, numSamples, model)
        else
            modelEstimates = testTrajectory(kinPoses, numSamples, model)
        end if
        Save raw kinematicPoses and modelEstimates
        allRawEstimates.append(modelEstimates)
        allRawOutputs.append(kinPoses)
        stats = calcPredictionStatistics(kinPoses, modelEstimates)
        individualStats.append(stats)
    end for
end for
Plot and save all individual statistics
    return calcPredictionStatistics(allRawOutputs, allRawEstimates)
```

investigated comprehensively in order to evaluate whether the NNs are able to learn a vehicle's motion model.

Utilising the learned model for localisation and mapping adds another level of complexity to the evaluation of the models. Specifically, the learned model's practical applicability needs to be evaluated, as well as the learned models' performance with regard to analytically defined models. The following section therefore provides a brief description of the parameters that need to be taken into account in order to reach a usable solution.

4.7.1 Parameter characterisation

Various parameters need to be taken into account to determine if the NNs are able to learn a vehicle's motion model. The learning rates, network structure, activation function, dataset and state memory can

have a large impact on the network's performance during estimation. Other factors that need to be taken into account include the spectrum of training data, the rate at which the model can provide estimates and the state's complexity and memory. In addition, different permutations of the aforementioned parameters can have an impact on the training results. For practical applications the noise in the transitional and measurement models can also affect the EKF's accuracy. Thus each of the parameters needs to be carefully evaluated using various configurations.

One example of how the parameters could influence learning is given by considering the network structure and the learning rates. For this test, the variable of interest is the NN's convergence/training error. Another would be the effect that the activation function type has on the dataset and how the function can affect the working interval or range. A third example is the number of previous states used versus the NN's structure. The focus of the test would be to establish if using more previous states requires more complex network structures in order to provide accurate estimates.

4.7.2 General test parameters

For the subsequent experiments various parameters were evaluated to find the most effective solution to model learning. Table 4.1 therefore provides the general setup parameters that were used to test the models' learning capabilities. Explanations for the landmark's parameters can be found in Addendum A. Any alterations or modifications to the parameters provided are specified in the actual experiments.

The purpose of the parameters listed are easily identifiable and interpretable, with a few exceptions. In general any parameters specified within block brackets (i.e. "[]") can be considered as a range of values or interval that the test requires, while a dash ("-") is used to indicate a NN with multiple hidden layers (e.g. $23-8$ has 23 neurons in the first hidden layer and eight in the second). Furthermore, any parameters specified in curly brackets (i.e. "{}") are parameters that were adjusted for different tests in order to find an optimal solution.

Table 4.1. General parameters used for the NN and EKF-SLAM implementations.

Parameter	**Value**	**Parameter**	**Value**
Algorithm	Back-propagation	Activation function	tanh
Weight range	$[-0.2, 0.2]$	Sampling rate	10 FPS (0.1s)
Maximum epochs	10000	Decimation indexes	$[0-29]$ (all)
Min training error	0.000001	Learning rate	0.12
Momentum	0.03	Annealing factor	[2000]
Number of previous states	3	Batch/Online processing	Online
NN structure	15	Scaling factor (x, y, θ)	[3.5, 3.5, 3.2]
Feature-detector	SIFT	Landmark increment	20
FLANN match ratio	0.6	Stability ratio	0.5
Group volume (Euclidean)	0.2m	Image memory (landmark creation)	6
Matching distance (Euclidean)	0.5m	Minimum matches within group	3
Training set	[Robot SLAM1, Robot SLAM2, Robot SLAM3]	Test set	Robot 360
Transitional noise (x, y, θ)	[0.00001, 0.00001, 0.00001]	Measurement noise (x, y, θ)	[0.05, 0.05, 0.05]
Initial covariance (x, y, θ)	[0.00025, 0.00025, 0.00025]		

4.8 SUMMARY

This chapter detailed the investigations into the training data, any manipulations required as well as the evaluation metrics used during the experimental results. The creation of simulated datasets was discussed, along with a methodology to evaluate the types of motion that could be learned. Lastly, the experimental procedures were described, along with the parameters of interest.

A framework was created to generate the simulated datasets automatically, apply the specified data manipulations to the training data and evaluate the various aspects of the learned models. The framework included providing summaries of training iterations, the statistical metrics used for the model learning evaluation methodology and the pose/trajectory results of the SLAM algorithms. Both the raw and processed results were stored for all the experiments. This includes the NN's weights, epoch errors, predictions for each tested trajectory and the raw trajectory of the EKF-SLAM algorithms.

CHAPTER 5 EXPERIMENTAL RESULTS

5.1 CHAPTER OVERVIEW

The following chapter details all the experimental protocols used to test the model learning strategy, as well as the outcome of each experiment. The chapter is divided into 3 main experiments: The real-world experiments using a tanh activation function (Section 5.2), using a linear activation function (Section 5.3) and determining the motion types that could be learned by the NNs (Section 5.4). Additional experimental results are also available in Addendum E.

For the initial tests only the real-world datasets with the primary approach (no control inputs added) were used to establish if a model could be learned. Subsequently, the impact of the linear activation function was evaluated to determine if any improvements could be observed. The performance of the NNs were compared to the analytical models after training to verify if any could be used to predict the vehicle's next state. Specifically, the accuracy of the models was quantified with respect to the ground-truth data. Both simulated landmarks and real-world data association were tested with various amounts of assumed noise for both the measurement and transitional model.

Lastly, the learning capability of the NNs were evaluated using the metrics described in Section 4.6. Both the real-world and simulated datasets were compared during these tests. For the simulated datasets both approaches described in Section 3.4 were used to train the models. The amount of training data required to learn a specific motion was also evaluated during the tests.

5.2 FREIBURG DATASET WITH TANH ACTIVATION FUNCTION

The initial tests conducted used the Freiburg datasets with no control input to train the NNs. The aim was to establishing the impact of various parameters during the learning process. The network structure, number of previous states and data format were considered the most important tests and were therefore evaluated first (see Section 4.2). Lastly, the NN's performance within a SLAM system was determined using the optimal configuration.

5.2.1 Experimental setup

As described by the parameters in Section 4.7.2, the NNs made use of the back-propagation algorithm with learning-rate, momentum and annealing factors. Three of the ground-truth datasets were used as training data where the poses were sub-sampled to a frame-rate of 10 FPS. Furthermore, decimation was employed between the indexes to increase the number of training examples. Various tests were conducted, each focusing on a specific parameter to find an optimal solution. Table 5.1 provides a summary of the parameters used during the subsequent tests.

The first test performed investigated the structure of the NNs using both single and two hidden-layer networks. While not all of the possible network structures were tested, the number of neurons used were distributed over relatively large interval to provide clear indications of the regions that yielded the best estimators. The expectation was that relatively small single layer networks (between 10 and 30 neurons) would provide sufficient interactions to represent the motion, with the learning rates, momentum and annealing factors reducing the oscillatory behaviour and the number of epochs required to reach convergence.

The amount of memory within the NNs was the third parameter investigated. For the memory tests three previous states were expected to provide sufficient information to learn second-order derivatives (acceleration) of the system, with additional memory introducing redundancies and potential learning of higher-order derivatives. It should be noted that since the number of inputs were altered during the memory tests, that the optimal NNs structure could differ from those found in previous tests. To account for such a possibility, a number of alternate network structures were also evaluated during the tests.

Table 5.1. Parameters for the initial NN tests.

Parameter	**Value**	**Parameter**	**Value**
Learning rate (l_r)	{0.45, 0.36, 0.3, 0.22, 0.12, 0.08 }	Number of previous states	{ 1, 2, 3, 5, 7, 9, 11, 13 }
Momentum (m)	{0.34, 0.29, 0.11, 0.03, 0.04, 0.03 }	Annealing factor (a_f)	{1000, 3000, 5000, 10000 }
NN structure	{62, 45, 33, 25, 19, 13, 7, 31-19, 24-9, 19-7, 17-11, 13-5, 7-5 }	Initial pose (x, y, θ)	$[-1.8478, 2.9448, 0.839]$
NN approach	Memory only, no control	Initial covariance (x, y, θ)	$[0.00025, 0.00025, 0.00025]$
Transitional noise (N_t)	$[0.01, 0.01, 0.01]$, $[0.001, 0.001, 0.001]$, $[0.005, 0.005, 0.005]$, $[0.0001, 0.0001, 0.0001]$, $[0.00001, 0.00001, 0.00001]$	Measurement noise (N_m)	$[0.1, 0.1, 0.1]$ $[0.05, 0.05, 0.05]$, $[0.001, 0.001, 0.001]$

Subsequently, the effects that data manipulation had on the NN's learning ability were investigated. The first manipulation investigated was splitting the datasets into smaller subsets whenever large discontinuities were observed between poses. While the NNs would not be able to provide accurate predictions whenever the shift-register contained discontinuities, the learning process would be simplified, leading to better representation of the motion. As with the data decimation, all of the discontinuous subsets were used as long as the subsets contained a minimum percentage of the total dataset (see Section 4.2.8). To ensure that no pattern was learned from the data, the order of training for the subsets were selected randomly during each epoch.

The second manipulation investigated incrementally added the datasets as described in Section 4.2.7. By incrementally adding the datasets after every n^{th} epoch, a significant speed-up in computation time should be observed while maintaining similar (or fewer) errors than when using the previous strategy. Furthermore, the NN's stability should also improve using the iterative approach, as more specific data was included initially. In addition, the discontinuity and iterative strategies were combined during

training and evaluated. The expectation was that if both strategies resulted in smaller errors, then a combination of the strategies would further improve the learned models.

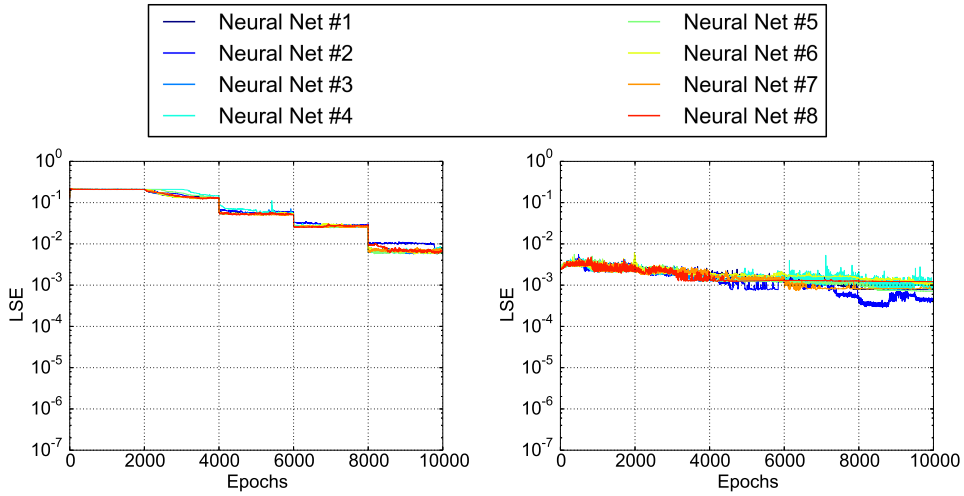
Lastly, the best performing NNs were used with the SLAM implementation to establish if the NNs could provide accurate estimates. For the initial tests, data association was not taken into consideration. Instead, simulated landmarks were used along with different transitional and measurement noises in order to gauge the effects on the predictions. The performance of the SLAM systems were compared to a generalised 2D odometry model, a differential drive's kinematic model and the ground-truth provided by the Freiburg datasets. If the correct model was learned, the outcome of the experiments were expected to improve predictions as more confidence was placed in the transitional model. In addition, the learned model's SLAM algorithm would also have smaller ATE and RPE errors than the analytical models.

5.2.2 Summary of results

5.2.2.1 NN structure and learning rate test results

Comparing the different NNs' structures LSE revealed that most single hidden-layer networks converged to a similar error (between $0.0015 - 0.0025$), with the smaller networks producing the smallest training errors. The largest difference between the different network sizes were the number of epochs required to reach a stable solution. The two hidden-layer networks, in comparison, showed that the errors either remained the same or increased as training progressed. Furthermore, none of the NNs could provide accurate estimates with the test vectors.

Changing the learning-rate, momentum and annealing factors also seemed to produce little improvement during training, with only slightly smaller errors observed ($0.00065 - 0.0011$). In addition the results showed that higher annealing factors required more epochs to reach the same error as lower annealing factors. In some cases the effects of higher annealing factors led to downward "steps" of the error. The reason the "step" were observed was due to annealing only being applied at certain intervals for learning rates and momentum, as discussed in Section 3.4.3. Figure 5.1 provides graphs showcasing typical errors observed during the aforementioned tests.



(a) $NN_{size} = 62$, $l_r = 0.18$, $M = 0.07$, $A_f = 2000$, $memory = 3$.

(b) $NN_{size} = 15$, $l_r = 0.12$, $M = 0.04$, $A_f = 10000$, $memory = 3$.

(c) $NN_{size} = 33$, $l_r = 0.12$, $M = 0.04$, $A_f = 2000$, $memory = 5$.

(d) $NN_{size} = 13-5$, $l_r = 0.12$, $M = 0.04$, $A_f = 2000$, $memory = 11$.

Figure 5.1. LSE error during training for the various NN structures.

5.2.2.2 NN memory test results

Increasing the size of the shift-register had two observable effects. For single hidden layer networks, an increase in memory brought about a slight increase in the error. Two hidden-layer networks, in comparison, brought about a slight decrease in errors. However, the training error of two hidden-layer networks was still larger when compared to single hidden-layer networks. Table 5.2 provides a summary of the training errors observed when the amount of memory was varied.

The test vectors for the NNs also revealed that large "jumps" between the estimates existed. Specifically,

Table 5.2. Range of NN training errors with regards to the amount of memory in the network.

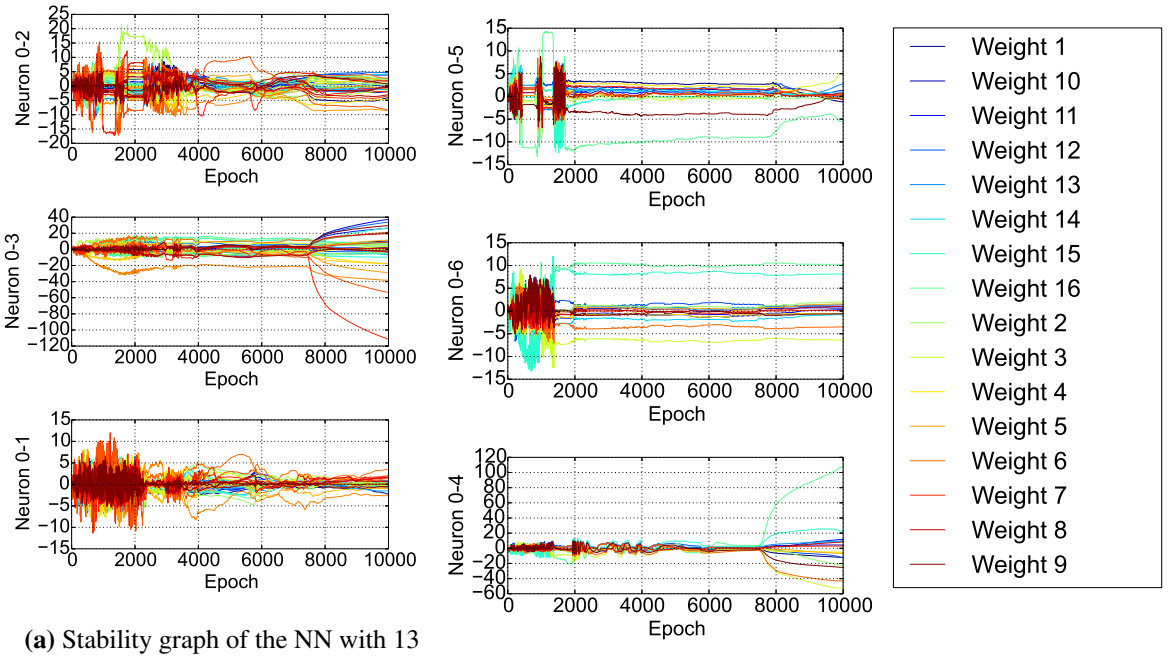
Memory	15 hidden-layer network	33 hidden-layer network	13-5 hidden-layer network
3	0.00026 - 0.00124	0.00079 - 0.00262	0.00206 - 0.01816
5	0.00082 - 0.00172	0.00122- 0.00235	0.00127 - 0.01651
7	0.00065 - 0.00171	0.00184 - 0.00270	0.00123 - 0.01749
9	0.00132 - 0.00179	0.00184 - 0.00296	0.001767 - 0.01726
11	0.00102 - 0.00185	0.00191 - 0.00291	0.00145 - 0.016297
13	0.00111 - 0.00195	0.00162 - 0.00386	0.001279 - 0.01173

many of the jumps occurred during the vehicle's orientation discontinuities. In Figure 5.2 the weights of the first three hidden neurons are graphed for the NN with five previous states as well as the fourth to sixth neuron's weights for the NN with thirteen previous states. In both cases, a significant increase in the weights were observed. For a tanh activation function such large values should not exist, as 98% of its range can be found in the interval $[-2.5, 2.5]$. The implication is that the NN may not be able to reach stable, convergent weight values.

Usually, when such large values are observed for a tanh activation function, then the input values are not scaled properly to the function's working interval. However, the problem was not with the scaled Freiburg datasets as the scaled data's maximum values were $[0.9419, 0.9220, 0.9810]$. Rather, the discontinuities appears to be the cause of the large weight values. The yaw output's neuron weights were consistently larger than the x- and y-outputs by a factor of 5, which also affected the hidden layer's neuron weights. Consequently, tests were conducted to determine if removing the discontinuities from the data allowed the NNs to converge to lower errors.

5.2.2.3 Discontinuity and iterative test results

Observations of the training results for the discontinuity, iterative and combined tests (Figure 5.3) show that removing the discontinuities reduces the LSE training error to between 4.5×10^{-6} and 7.2×10^{-6}. The results therefore indicate that NNs have difficulty learning how to handle discontinuous data, especially in scenarios where such discontinuities occur infrequently.



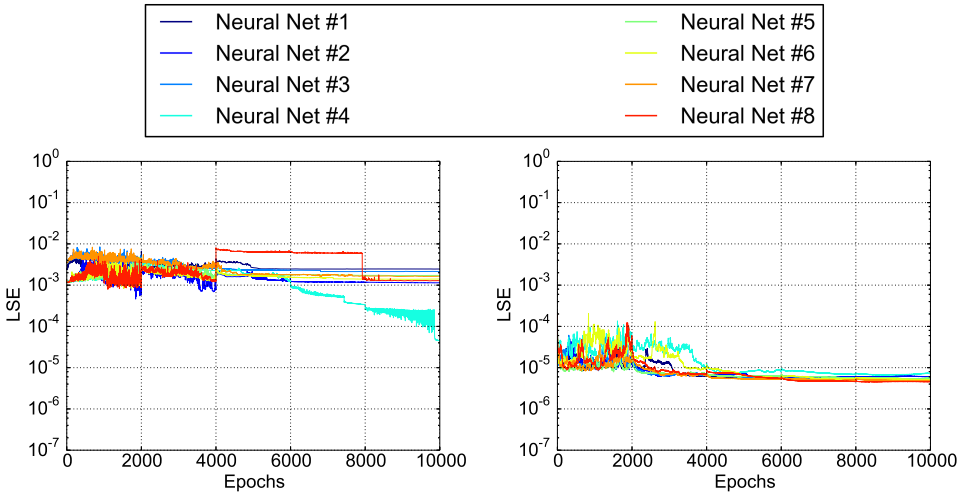
(a) Stability graph of the NN with 13 previous states.

(b) Stability graph of the NN with 5 previous states.

Figure 5.2. Stability graph of the NN's hidden layer (partial) for a NN with 15 hidden nodes.

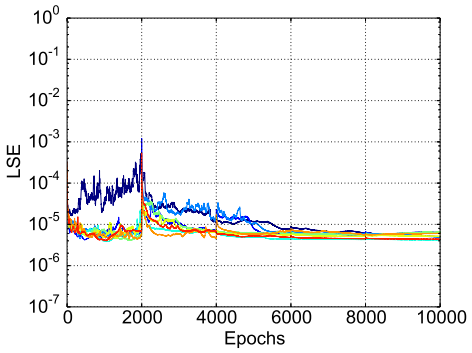
In comparison, the iterative tests show almost no improvement over previous results except in rare instances where one of the NNs outperformed the rest (an LSE of 4.5×10^{-5}). Combining the discontinuity and iterative tests also confirmed that the iterative tests did not affect the overall performance. In fact, comparing the combined tests with the discontinuous tests revealed that the combined tests performed slightly better on average (LSE between 4.1×10^{-6} and 6.8×10^{-6}).

The test vectors (Figure 5.4) also confirm the result of the training error. Iteratively appending the datasets offers no real improvement on the estimates, while splitting the data at discontinuities significantly reduces the errors except when the discontinuities were inside the shift register. Furthermore, the combined tests produced similar results to the discontinuous tests, with some slight improvement at the discontinuities. Lastly, examining the stability of the NNs revealed that removing the discontinuities significantly lowered the overall weights in the NNs. Specifically, the weights for the iterative tests were regularly an order of magnitude higher than the NNs trained with the discontinuities removed.



(a) Training error where the datasets are added iteratively.

(b) Training error with the discontinuities removed from the input data.



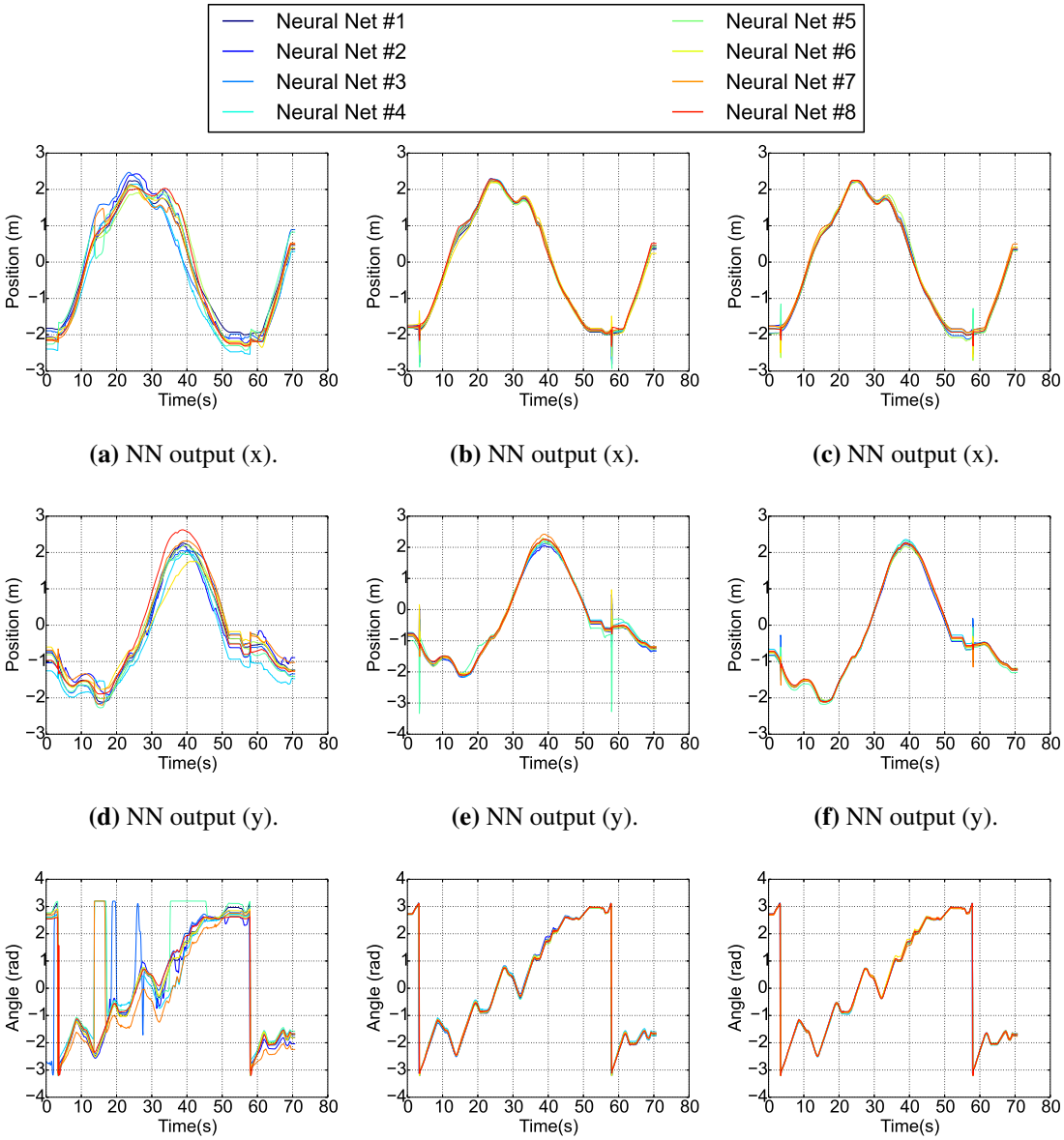
(c) Training error where discontinuities are removed and datasets added iteratively.

Figure 5.3. LSE error during training for the discontinuities and iterative tests.

5.2.2.4 SLAM with simulated landmarks

The results of the simulated SLAM experiments are given in Figure 5.5. Two observations can be made with regard to the amount of noise on the system. The first is that the NN's pose estimates tend to the previous estimate as the transitional noise decreases. Secondly, the yaw estimates tend to a constant value as the transitional noise decreases.

The ATE results also concur with the observations that a lower transitional noise leads to larger errors during estimation (see Table 5.3 and Figure 5.6). Further investigation into possible causes revealed



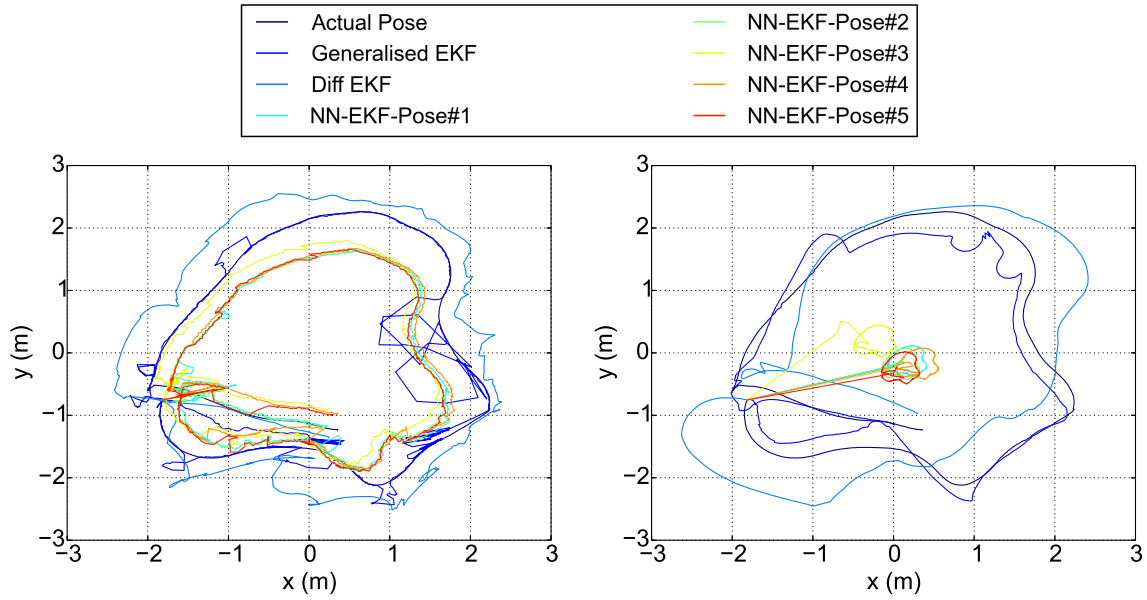
(a) NN output (x). **(b)** NN output (x). **(c)** NN output (x).

(d) NN output (y). **(e)** NN output (y). **(f)** NN output (y).

(g) NN output (θ). **(h)** NN output (θ). **(i)** NN output (θ).

Figure 5.4. Plot of the test vector's results compared to the actual output for the iterative tests (left) discontinuous tests (middle) and the combined tests (right).

that the actual learned weights may be the reason for the estimates. In almost all the tests conducted the bias variable's weight had a significant impact on the final estimate. Thus, a likely explanation is that the NNs were unable to learn the inherent model, as a significant portion of the final estimate was not dependent on any of the previous states. One possible reason for the NN learning to add significant weight to the bias variable is that the activation function is unsuitable for motion model



(a) Pose-graph of the NN-EKF with $N_t = 0.01$ and $N_m = 0.05$.

(b) Pose-graph of the NN-EKF with $N_t = 0.0001$ and $N_m = 0.1$.

Figure 5.5. Pose graphs of the NN-EKF with different transitional and measurement noise parameters with a NN structure of 15 hidden nodes.

learning. Because the tanh activation function adds an incremental gradient between two extremes (either -1 or 1), an activation function that is unconstrained may be able to encapsulate the motion model more accurately.

5.3 FREIBURG DATASETS (LINEAR ACTIVATIONS)

The previous experiments demonstrated that the NNs were highly inconsistent during localisation and mapping. As such the NNs could not be used to predict or update the vehicle state accurately. The following experiment therefore determined the effects that a linear activation function had on the learning process. As with the previous experiments, no control input was added to the training data.

Table 5.3. ATE results for the different EKF-SLAM simulations.

	$N_t = 0.01, N_m = 0.05$		$N_t = 0.0001, N_m = 0.1$	
Algorithm	**RMSE**	σ	**RMSE**	σ
Odometry EKF-SLAM	0.2791	0.1312	0.4040	0.1341
Differential drive EKF-SLAM	0.6300	0.2659	0.3751	0.89116
NN EKF-SLAM 1	0.2106	0.1019	1.3782	0.3502
NN EKF-SLAM 2	0.1748	0.0771	1.3035	0.3302
NN EKF-SLAM 3	0.2576	0.0985	1.4751	0.3787
NN EKF-SLAM 4	0.2834	0.1042	1.4325	0.3634
NN EKF-SLAM 5	0.2045	0.0807	1.3065	0.3315

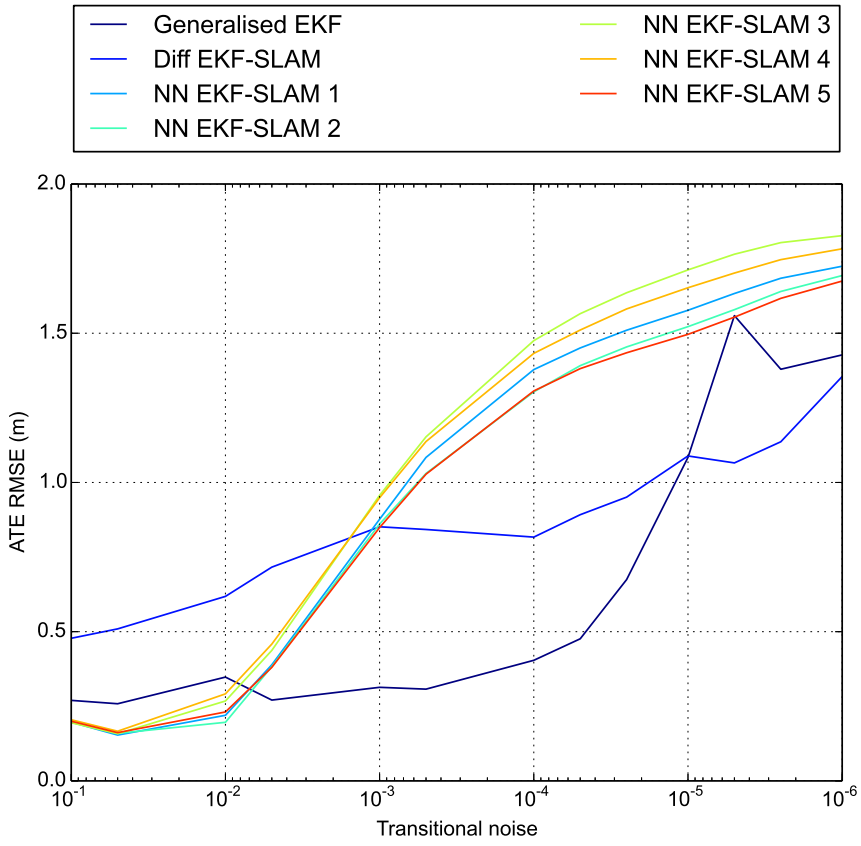


Figure 5.6. Graph of the transitional noise vs. the RMSE of the ATE metric where measurement noise was assumed to be $N_m = 0.05$.

5.3.1 Experimental setup

Changing the activation function influences the learning process significantly and as such most of the parameters evaluated previously were re-evaluated. In particular, the NN's may require additional neurons and layers to learn any non-linearities inherent in the motion. Hence changes in the NN's structure was re-evaluated during the experiments. While a memory of three was still expected to be sufficient to learn the higher-order dynamics, tests were conducted for verification and to ensure consistency.

Factors that were not specifically re-evaluated included the learning rate, momentum and annealing factors. Instead, the values found in the previous experiments were used. Similarly, splitting the datasets at discontinuities and adding the datasets iteratively were also not re-evaluated. Instead, both formats were included from the onset, as this was found to decrease the training errors significantly. To eliminate any bias, the training and test sets were also changed for the experiments, as shown in Table 5.4.

The best performing NNs were used within the EKF-SLAM system to determine the performance of localisation and mapping. The SLAM implementation made use of the simulated landmarks with the transitional and measurement noises varied to establish the noise's effects. As a final test, the data association problem was taken into consideration to produce a full SLAM implementation. Specifically, the landmarks were created from data received from a Xbox Kinect and updated as described in Addendum A. The results were again compared to a generalised 2D odometry model, a differential drive's kinematic model and the ground-truth provided by the Freiburg datasets.

5.3.2 Summary of results

5.3.2.1 Linear activation function training results

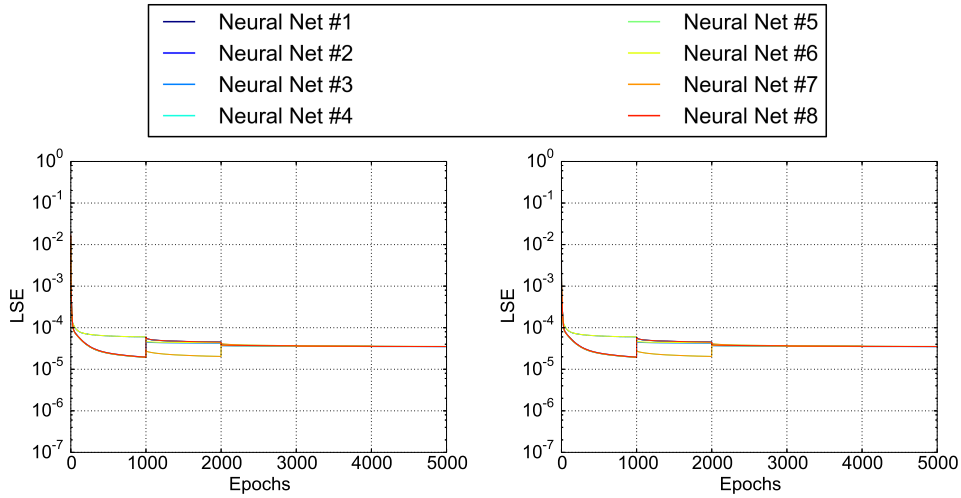
Observations of the training results appeared to reach stable minimums. Both single- and two-layer networks converged to a training error of approximately 3.74×10^{-5}, as shown in Figure 5.7. In addition, the NN's structure did not appear to have a large impact on the training error, with all the tested NNs producing similar errors.

Table 5.4. Parameters for the linear activation function tests.

Parameter	**Value**	**Parameter**	**Value**
NN structure	{ 15, 28, 36, 43, 13-7, 24-11 }	Activation functions	Linear
Discontinuity split	Yes	NN approach	Memory only, no control
Training set	[Robot SLAM1, Robot SLAM2, Robot 360]	Test set	Robot SLAM3
Number of previous states	{ 1, 2, 3, 5, 7, 9, 11, 13 }	Image memory	3
Initial covariance (x, y, θ)	[0.00025, 0.00025, 0.00025]	Initial pose (x, y, θ)	$[-1.8478, 2.9448, 0.839]$
Transitional noise (N_t)	$[0.01, 0.01, 0.01]$, $[0.001, 0.001, 0.001]$, $[0.005, 0.005, 0.005]$, $[0.0001, 0.0001, 0.0001]$, $[0.00001, 0.00001, 0.00001]$	Measurement noise (N_m)	$[0.1, 0.1, 0.1]$ $[0.05, 0.05, 0.05]$, $[0.001, 0.001, 0.001]$

The amount of memory did not have a significant impact on the training error either. The only exception was when no memory was added to the NN. In that case, the errors were approximately 1.9×10^{-4}. The tests vectors, however, did reveal that using more memory resulted in larger oscillations around the yaw discontinuities. The cause of these oscillations were due to the discontinuity remaining longer in the NN's memory. Furthermore, the NN's weights reached stable values in fewer epochs and tended to remain at those values when compared to the tanh activation functions (see Figure 5.8). Comparisons of the weights with the tanh activation function's NNs also revealed that the bias variable's weight no longer had a large influence on the estimates. Instead, the prediction of the next state mostly depended on the previous state, with the memory contributing less weight to the prediction.

The only concerning result from the linear activation functions was that the training errors were approximately one order of magnitude higher than the NNs with tanh activations. One explanation for the higher error is that linear activation functions do not perform as well as tanh activation functions when using gradient descent algorithms. An alternate explanation is that the NNs with tanh activation never



(a) Training error with a *linear* activation function, 15 hidden nodes and three previous states.

(b) Training error with a *linear* activation function, 15 hidden nodes and 13 previous states.

(c) Training error with a *linear* activation function and 43 hidden nodes and three previous states.

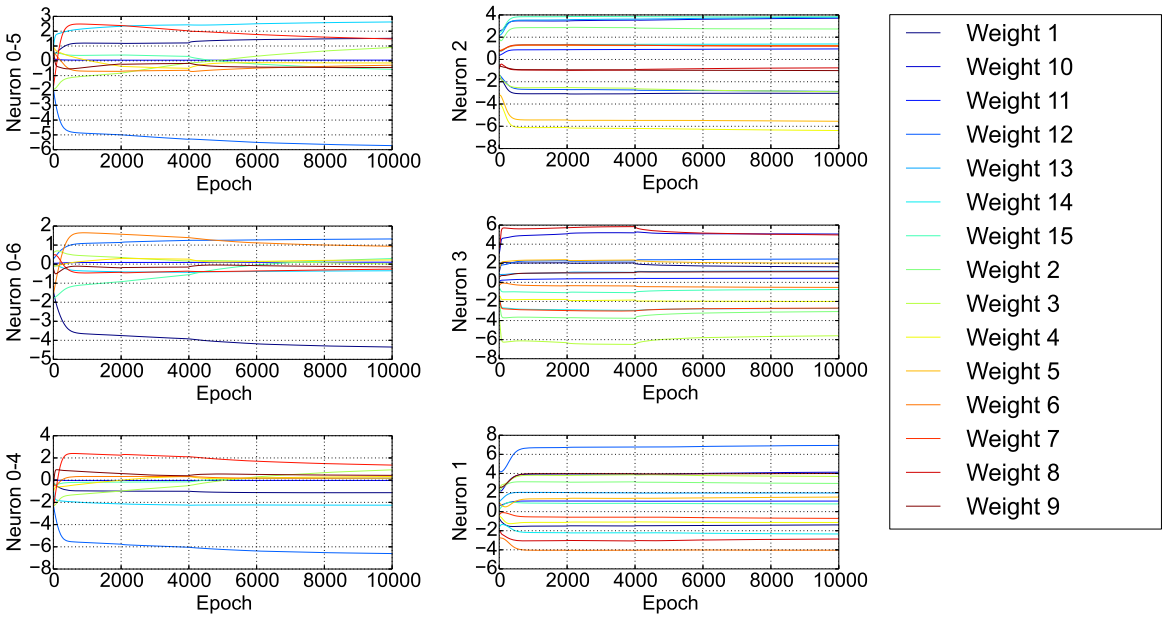
(d) Training error with a *linear* activation function, 13-5 hidden nodes and 11 previous states.

Figure 5.7. LSE error during training for the discontinuities and iterative tests.

learned the model, which is why the bias variable had a significant influence to the prediction.

5.3.2.2 SLAM with simulated landmarks

The simulation results obtained from the linear activation functions showed a marked improvement over the previous results. Specifically, the warping observed with the tanh activation function was not observed and the yaw estimates did not converge to some constant value. Instead, the trajectories



(a) Hidden weights for neuron 5-8. **(b)** Output weights.

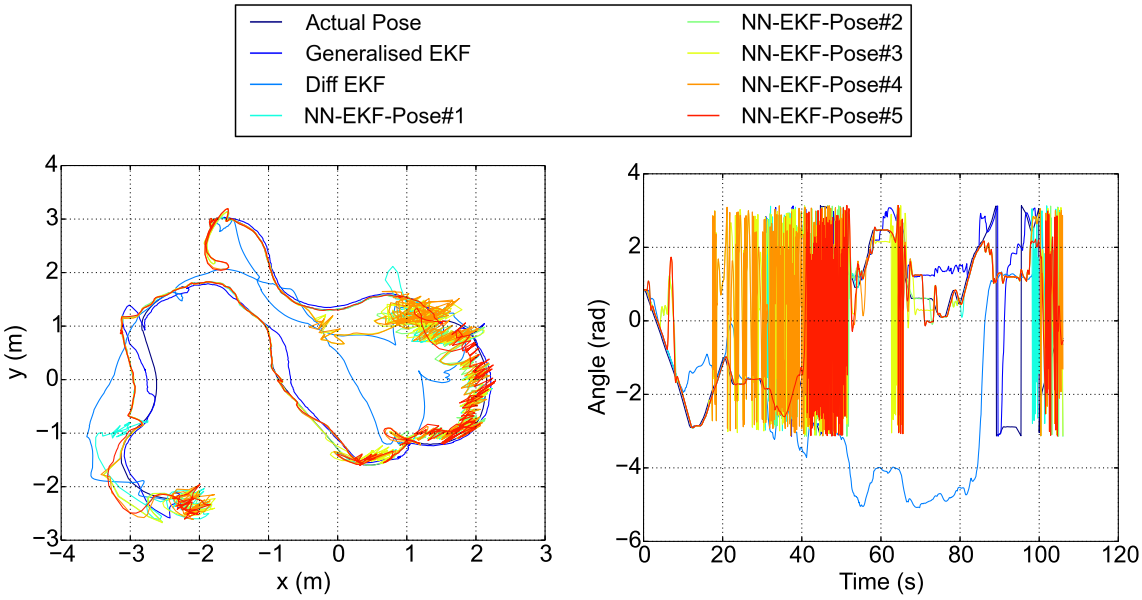
Figure 5.8. Stability graph of the NN's hidden layer (partial) and output layer for a NN with a hidden node structure of 15 and a memory of 3.

tended to follow the vehicle's trajectory, with some oscillations at certain poses.

These oscillations were more pronounced when a lower transitional noise was used with the tests, as shown in Figure 5.9 and Figure 5.10. Furthermore, the oscillations occurred most frequently during phase-changes, which indicates that the oscillations may have been caused by discontinuities in the NN's memory.

The observations of the different NN-EKF simulations indicate that a number of problems still exist during estimation. The first one is the oscillations occurring at the phase-changes. As the phase-changes are the primary cause of the oscillations, the best solution would be to ensure that the NNs never observe the phase-change. By allowing the yaw estimates to be unconstrained, the NNs will never observe the phase changes. Once the full trajectory has been observed, the yaw estimates can then just be re-factored to the interval $[-\pi, \pi]$. Hence the strategy was implemented and evaluated in the subsequent section's experiments.

The second problem is that lower transitional noises do not result in lower ATE errors. Thus learned



(a) Pose-graph of the NN-EKF with $N_t = 0.001$ and $N_m = 0.05$.

(b) Yaw of the NN-EKF with $N_t = 0.001$ and $N_m = 0.05$.

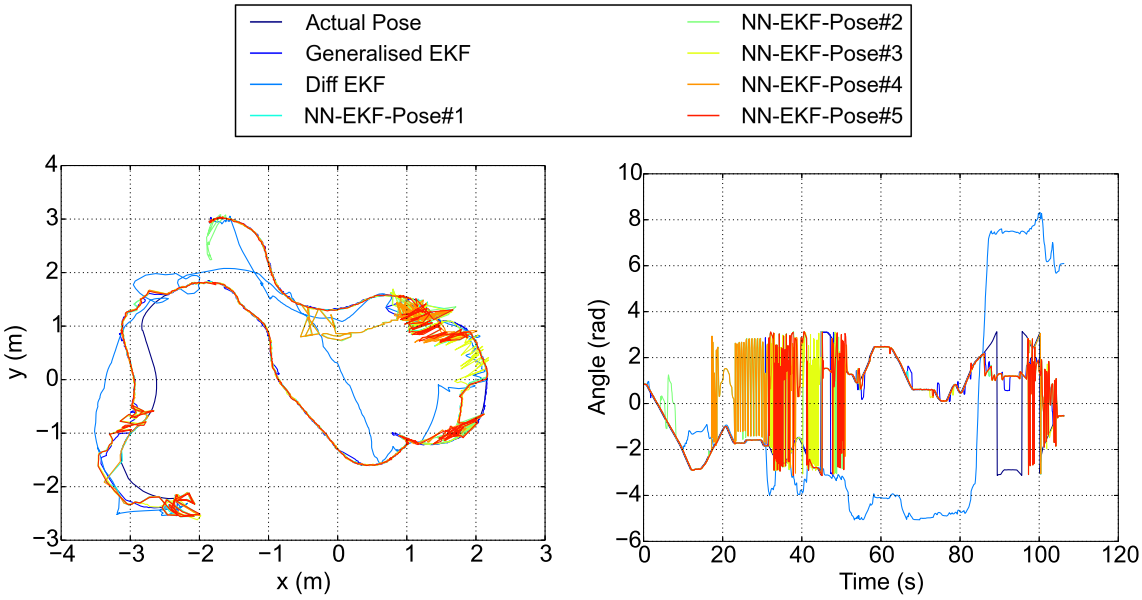
Figure 5.9. Pose graph of the NN-EKF using a NN with 15 hidden nodes.

models may not provide accurate predictions to the system. Instead the measurement model appears to compensate the incorrect predictions. Furthermore, the compensation occurs more readily when a larger noise is assumed for the transitional model. Data association was therefore taken into consideration to determine whether the learned models provide similar real-world results.

5.3.2.3 SLAM with data association

The results obtained for the NN-EKF confirmed that the NNs were unable to provide accurate predictions of the vehicle's motion, as shown in Figure 5.11. While the yaw estimates no longer contained the oscillations seen with the simulated landmarks, the predicted trajectory still did not resemble the ground-truth.

The RPE and ATE results further demonstrated that the learned models could not provide accurate predictions to the NNs. Even though the errors for the differential drive and odometry models were also high because of misalignments in the trajectory, both still yielded smaller RPE and ATE errors



(a) Pose-graph of the NN-EKF with $N_t = 0.01$ and $N_m = 0.05$.

(b) Yaw of the NN-EKF with $N_t = 0.01$ and $N_m = 0.05$.

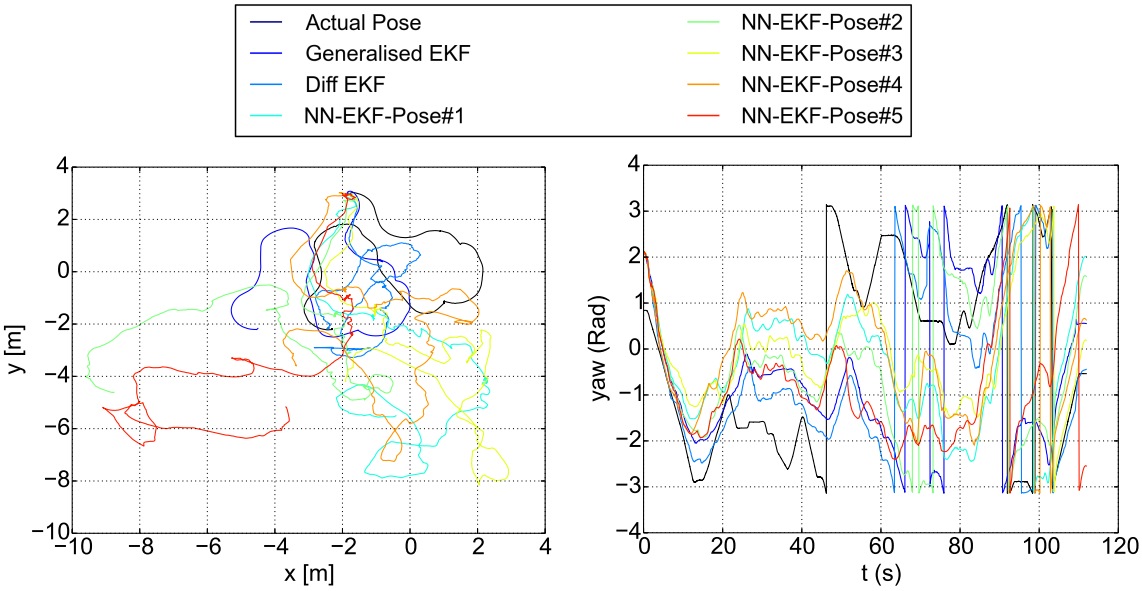
Figure 5.10. Pose graph of the NN-EKF using a NN with 13-7 hidden nodes.

than the learned models, as shown in Table 5.5. In addition, the standard deviation and maximum error over the trajectory were generally also lower for both analytical models.

Further tests were also conducted using the Robot 360 dataset (a training set) to ascertain whether the NNs learned the specific trajectories instead of a general model. However, the results remained similar to the ones obtained using the SLAM3 dataset. Further investigation was therefore needed to determine if the NNs could learn even the most simplistic motion.

5.4 MODEL LEARNING TESTS

The previous experiments demonstrated that the NNs could not learn a vehicle's motion model sufficiently to provide accurate predictions. However, no information was obtained as to why the NNs could not properly learn the vehicle models. Furthermore, questions on the types of motion that could be learned, if any, also required investigation.



(a) Pose-graph for the NN with 15 hidden nodes and a memory of three.

(b) Yaw-graph for the NN with 15 hidden nodes and a memory of three.

Figure 5.11. Pose-graphs of the NN EKF-SLAM with the $N_t = 0.001$ and $N_m = 0.05$.

Table 5.5. RPE results for the different SLAM simulations with $N_t = 0.001$ and $N_m = 0.05$.

Algorithm	**Translational component**		**Rotational component**	
	RMSE (m)	$\sigma(m)$	RMSE (°)	$\sigma(°)$
Odometry EKF-SLAM	1.3190	0.5484	80.6468	37.1605
Differential drive EKF-SLAM	1.2627	0.5211	78.9587	35.4953
NN EKF-SLAM 1	1.6309	0.6773	85.8827	44.8140
NN EKF-SLAM 2	1.5766	0.7047	83.5064	40.6333
NN EKF-SLAM 3	1.6443	0.7637	83.6391	41.3253
NN EKF-SLAM 4	1.5607	0.6789	91.0661	44.7849
NN EKF-SLAM 5	2.2952	1.4413	86.9623	43.4606

5.4.1 Experimental setup

Determining the types of motion that the NNs are able to learn requires that each type be isolated from the others during training. As discussed in Section 4.3, simulated datasets can be used to easily create

separable datasets for different motion types. Hence the subsequent experiments trained each motion type both separately and as a combined set. The various setup parameters used during these tests are described in Table 5.6.

Table 5.6. Parameters for the linear activation function tests.

Parameter	**Value**	**Parameter**	**Value**
NN structure	{3, 7, 11, 25, 33, 45, 150, 11-5, 23-9 }	Activation functions	linear
Discontinuity split	Yes	Test set	Simulated test sets
Training sets (simulated)	[Arc, linear, rotational, stationary]	NN approach	[Memory, memory with control]
Number of previous states	{ 1, 3, 7, 11 }	Increasing training data	[Initial poses, control trajectories]

The learned models were evaluated following the methodology specified in Section 4.6. Thus each trajectory was evaluated using the metrics defined and a summary of each control and subsequent motion type was made. For completeness, the models learned from the Freiburg datasets were evaluated initially to provide a baseline for comparison. In addition, the influence that the amount of training data had on each motion type was investigated. Specifically, increasing the number of initial poses and controls included in a training set was expected to increase the accuracy of the learned models. The NN's structure and memory were also re-evaluated for the experiments to determine if the simulated trajectories impacted the parameters previously found during training.

Lastly, the alternative approach as described in Section 3.4.2 was investigated. Hence the vehicle control used to generate each trajectory was added as input to the NNs and used during training. Comparisons were drawn between the NNs with and without control, where the NNs with control were expected to outperform the NNs without control. The network structure, memory and diversity of the training data used during training were again evaluated in order to cover the problem space comprehensively.

5.4.2 Summary of results

Analysis of the metrics for the overall motion types and individual trajectories revealed that some metrics were more suitable than others. Specifically, the AE_{min} and AE_{max} only provided limited information on the errors, while the MSE errors differed significantly from the MAE and AE_{median}, especially at the yaw estimates. Similarly, the standard deviation of the yaw error did not correspond with the median absolute deviation ($AE_{medianAD}$).

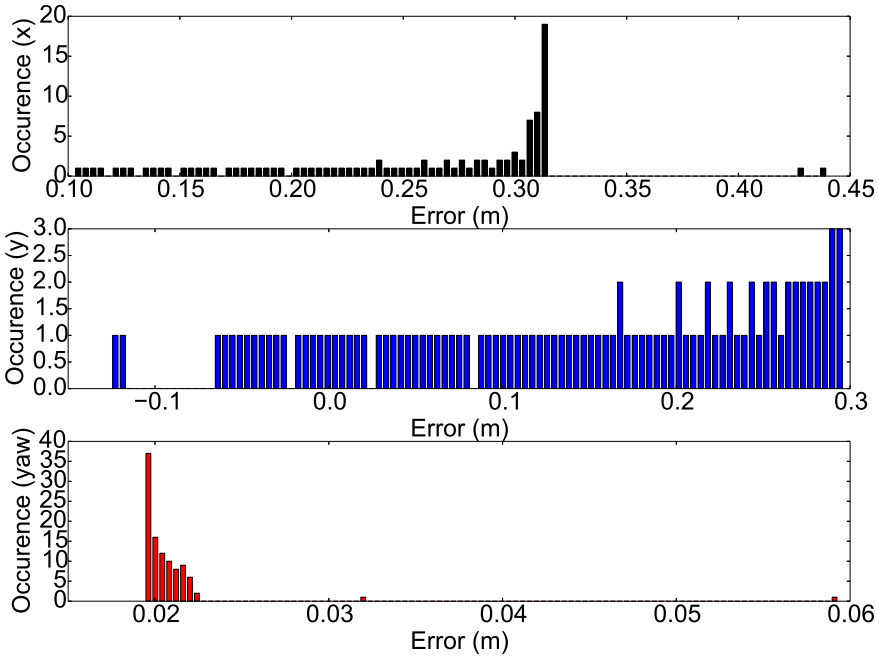
In both cases, the disagreements of the metrics were due to the yaw discontinuities. Lastly, the large kurtosis and skew values observed individually indicated that any mean value might not effectively represent the probability distribution of the errors. Therefore, the AE_{median}, $AE_{medianAD}$, AE_{skew} and $AE_{kurtosis}$ were the preferred metrics used during evaluation.

5.4.2.1 Freiburg dataset results

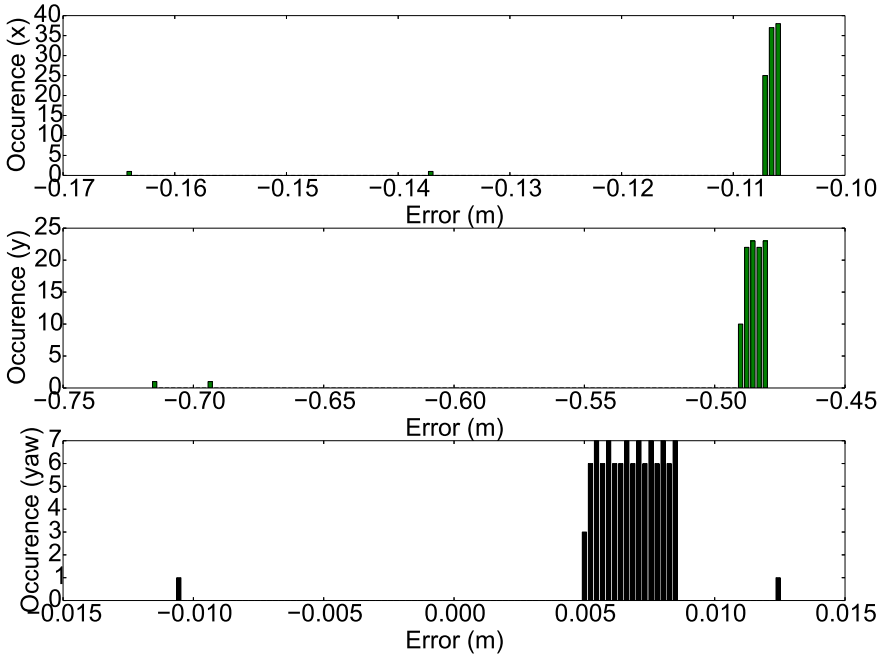
The various tests conducted on the Freiburg datasets revealed that the type of motion tested produced different errors. Histograms of the errors for each motion type were therefore investigated. Typical results for each type are provided in Figure 5.12 and Figure 5.13 , where histograms in green indicate that a large skew was detected (larger than $|4|$) while blue was used when a kurtosis less than -1.4 was detected. Histograms in red were used when both a large skew and small kurtosis were detected.

For arc motion, skew values exceeding either -4 or 4 were regularly observed for the yaw estimates with large excess kurtosis values. Small values for the kurtosis were also common for the x- and y estimates during the predictions, leading to a probability distribution that appeared to be uniformly distributed over the error interval. Linear motion, in comparison, regularly had large skew and kurtosis values for both the x- and y- estimates, while the yaw estimates were prone to vary, depending on the initial pose and control.

Pure rotational motion showed that the x-estimates generally had small skew and kurtosis values, while the yaw estimates had large skew and kurtosis values. The y-predictions for rotational motion, however, produced both skew and kurtosis values that varied not only over different trajectories but also over training iterations of the NNs. Tests using stationary motion provided a number of noteworthy

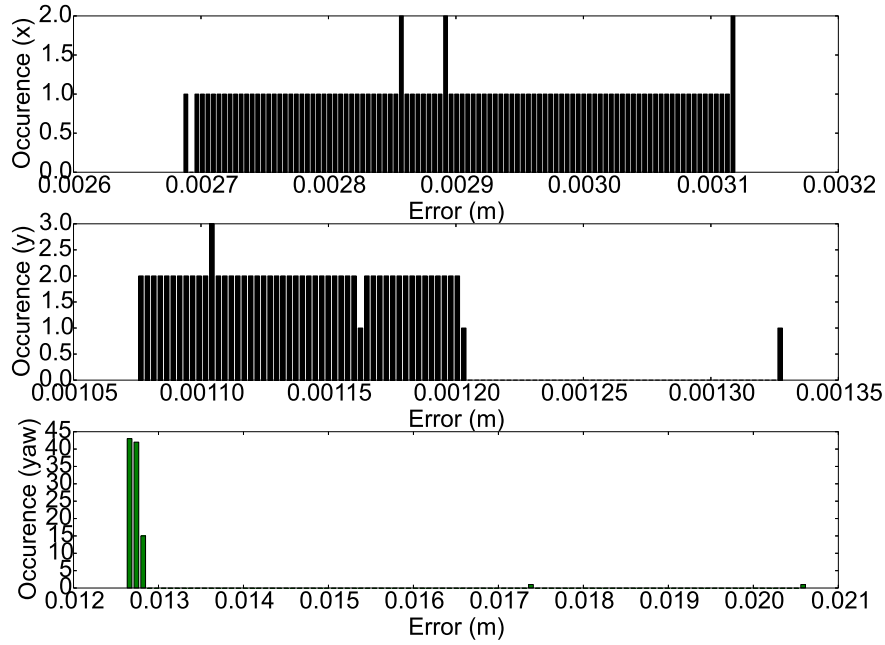


(a) $P_{init} = -0.6, -1.7, -0.3$, $v_f = 1.5$, $\omega = 0.15$.

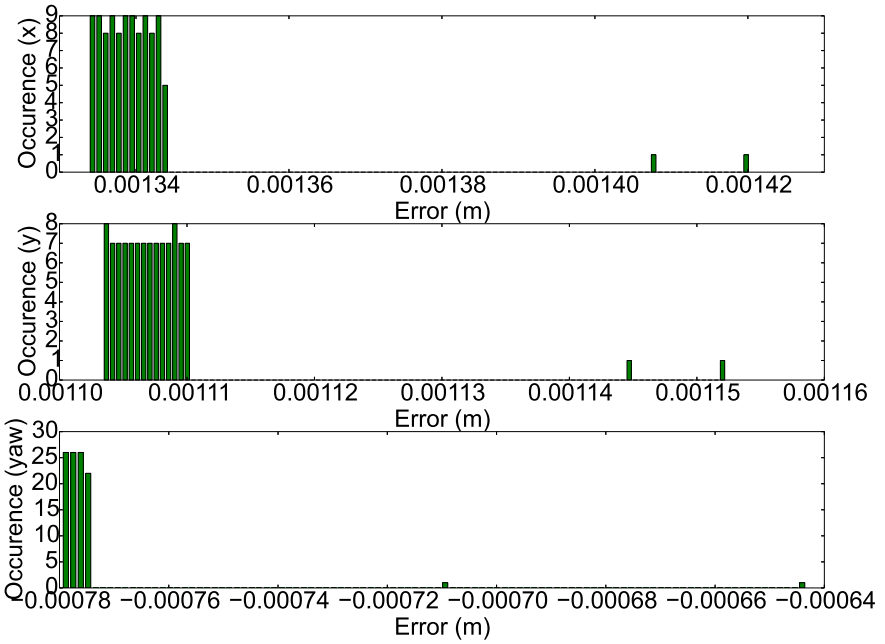


(b) $P_{init} = 0.8, -1.2, -1.78$, $v_f = 2.4$, $\omega = 0.0$.

Figure 5.12. Histograms for the NN with 15 hidden nodes and a memory of three. P_{init} is the initial pose, v_f the forward velocity and ω the rotational velocity used to generate the test trajectory.



(a) $P_{init} = 2.4, 1.3, 2.4$, $v_f = 0.0$, $\omega = -0.07$.



(b) $P_{init} = 1.2, 1.2, 0.5$, $v_f = 0.001$, $\omega = 0.001$.

Figure 5.13. Histograms for the NN with 15 hidden nodes and a memory of three. P_{init} is the initial pose, v_f the forward velocity and ω the rotational velocity used to generate the test trajectory.

observations. When the forward velocity was set to 0.0, the skew and kurtosis for the x-estimates remained close to zero. However, as soon as a small amount of forward velocity was provided (e.g.

0.001), large skew and kurtosis were observed. The y-predictions provided similar results to the rotational motion when using no forward velocity, with large kurtosis and skew values when a small forward velocity was provided. For the yaw estimates large skew and kurtosis values were observed irrespective of the forward velocity supplied.

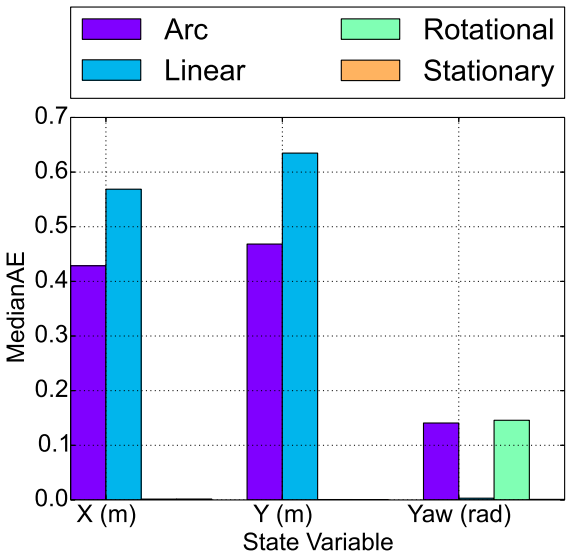
The overall errors for each motion type were determined and graphed as shown in Figure 5.14. From the AE_{median} graph one can observe that linear motion produced the largest x- and y errors, while arc and rotational motion produced the largest yaw errors. Furthermore, each motion type except for stationary motion were subject to large deviations over the tested trajectories. The NNs could therefore only be used to represent stationary motion.

Further analysis of different NNs revealed that increasing the memory significantly reduced the overall performance. From Table 5.7 one can observe that less memory provides better estimates. However, even with a memory of 3 the x-, y- and yaw errors were significant, which explains why the NNs could not provide accurate estimates when executing the EKF-SLAM algorithm. In addition, the NNs that contained no memory were able to provide better estimates than those containing memory. Hence the results for the NNs' memory contradict the training errors observed in previous experiments, which could indicate that the NNs cannot learn higher-order dynamics.

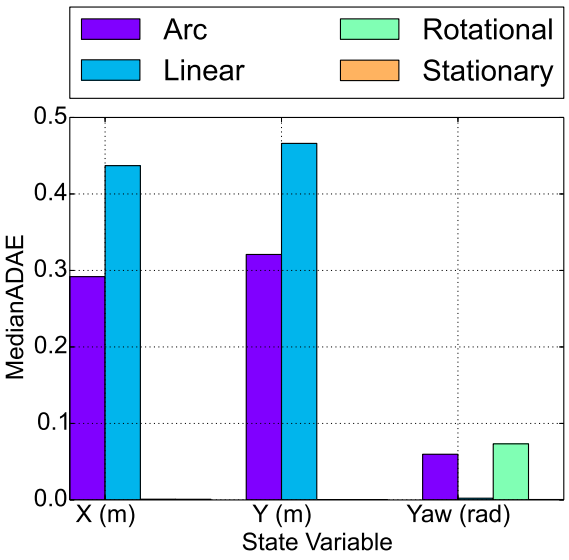
5.4.2.2 Network structure tests (simulated datasets)

The first tests conducted with the simulated datasets were intended to establish whether the network structure had any impact on the models' performance. Furthermore, each motion type was trained separately on the various NNs to determine if a specific type of motion could be learned. The number of initial poses used by the training data was set to 15, with the arc motion control set to 10. For the linear and rotational motion the control was set to eight, while stationary motion made use of six controls (see Section F.2). In addition, each motion type was evaluated after training, irrespective of the training data type. However, only the relevant test type's results are discussed in this section.

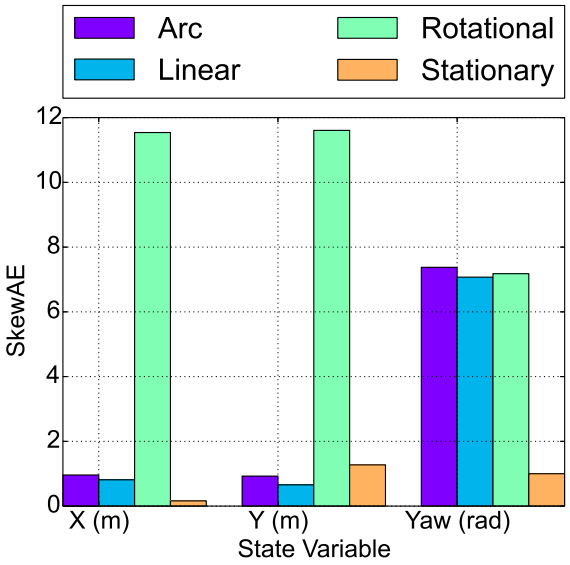
The results observed showed that all the NNs with two hidden layers could not train properly. Consequently, the two hidden layer networks' results are not discussed further. Table 5.8 and Table 5.9 provide the overall results for training the NNs with varying amounts of neurons, with and without



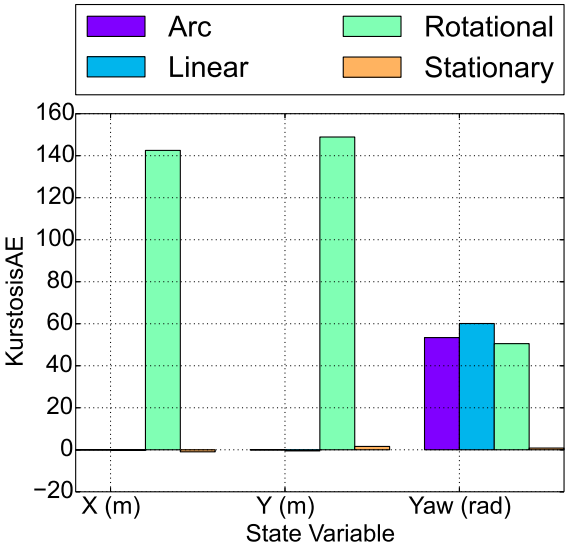
(a) AE_{median}.



(b) $AE_{medianAD}$.



(c) AE_{skew}.



(d) $AE_{kurtosis}$.

Figure 5.14. Bar graphs of each motion type's error for a NN with 15 hidden nodes and a memory of 11. The x-, y- and yaw errors were each graphed next to each other for direct comparison.

control. As in the previous tests conducted using linear activations, the number of hidden neurons did not seem to have a large impact on the estimates.

The type of motion used, however, did have a significant impact on the results. Predictions for stationary motion without control input could be represented with a fair degree of accuracy. Furthermore,

Table 5.7. The overall AE_{median} results for the NNs trained with Freiburg datasets using various amounts of memory.

	Memory						
Test Data	**1**	**3**	**5**	**7**	**11**	**13**	**15**
Arc	0.047	0.089	0.174	0.259	0.425	0.51	0.595
motion	0.047	0.096	0.189	0.283	0.467	0.56	0.651
(x,y,θ)	0.015	0.0259	0.053	0.082	0.14	0.17	0.193
Linear	0.062	0.118	0.229	0.342	0.567	0.68	0.794
motion	0.066	0.127	0.253	0.39	0.633	0.76	0.884
(x,y,θ)	0.0078	0.0055	0.0033	0.003	0.0035	0.0044	0.0053
Rotational	0.0092	0.0011	0.00095	0.001	0.0014	0.00145	0.00155
motion	0.0033	0.00045	0.00058	0.00063	0.0007	0.001	0.0008
(x,y,θ)	0.013	0.028	0.058	0.089	0.145	0.176	0.2
Stationary	0.0092	0.00105	0.001	0.001	0.0015	0.0013	0.0014
motion	0.0029	0.00045	0.004	0.0005	0.0006	0.0009	0.00085
(x,y,θ)	0.0027	0.00075	0.00065	0.001	0.001	0.0014	0.0016

any variables that remained constant also had low prediction errors, as indicated by the linear and rotational motion's results. However, the errors for any variables that required some interaction could not be properly represented using the original approach with a median error of up to 0.128m per prediction.

By including the control variables as input (Table 5.9), a significant improvement was observed. In particular, the NNs were able to provide reasonably accurate estimates for the special case motions. However, the arc motion errors were generally an order of magnitude higher than the special case motions, with errors reaching as high as 0.018m for both the x- and y estimates over certain trajectories.

Table 5.8. The overall AE_{median} error (measured in meters and radians) for the NNs using various NN structures with a memory of 3.

	NN structure					
	3	**7**	**11**	**33**	**45**	**150**
Test Data	**Training data: Arc motion**					
Arc motion	0.085	0.085	0.087	0.086	0.087	0.085
(x,y,θ)	0.095	0.095	0.095	0.094	0.0945	0.094
	0.029	0.029	0.029	0.029	0.029	0.030
	Training data: Linear motion					
Linear	0.115	0.115	0.113	0.115	0.112	0.115
motion	0.128	0.126	0.126	0.125	0.126	0.128
(x,y,θ)	0.00018	0.00018	0.00019	0.00019	0.00019	0.00018
	Training data: Rotational motion					
Rotational	2.40×10^{-6}	2.38×10^{-6}	2.40×10^{-6}	2.40×10^{-6}	2.38×10^{-6}	2.32×10^{-6}
motion	2.0×10^{-6}	1.90×10^{-6}	1.90×10^{-6}	1.85×10^{-6}	1.78×10^{-6}	1.74×10^{-6}
(x,y,θ)	0.0299	0.0299	0.0299	0.0299	0.0299	0.0299
	Training data: Stationary motion					
Stationary	0.00011	8.9×10^{-5}	9.9×10^{-5}	9.6×10^{-5}	8.8×10^{-5}	8.8×10^{-5}
motion	7.1×10^{-5}	7.6×10^{-5}	9.6×10^{-5}	7.9×10^{-5}	6.5×10^{-5}	7.1×10^{-5}
(x,y,θ)	0.00028	0.00031	0.00025	0.00028	0.00032	0.00026

5.4.2.3 Memory tests

The effects that the amount of memory had on the simulated datasets' performance were also investigated using both approaches, as shown in Table 5.10. Arc motion was the primary focus of the tests because arc motion was considered the general vehicle motion. The results for the NNs without control input were found to be similar to the Freiburg datasets. However, when control was included and the NNs contained no memory, the prediction accuracy decreased for the x-estimates while increasing for the yaw estimates. Thus, simply adding control without memory did not allow the NNs to learn the motion.

Table 5.9. The overall AE_{median} error (measured in meters and radians) for the NNs using various NN structures with control input included and a memory of 3.

	NN structure					
	3	**7**	**11**	**33**	**45**	**150**
Test Data	**Training data: Arc motion**					
Arc	0.0016	0.0021	0.0019	0.002	0.0019	0.0017
motion	0.0019	0.0018	0.0019	0.0023	0.0019	0.0018
(x,y,θ)	0.0013	0.0012	0.0012	0.0013	0.0012	0.0013
	Training data: Linear motion					
Linear	0.0006	0.0005	0.0006	0.0006	0.0006	0.0005
motion	0.0003	0.0003	0.0003	0.0005	0.0005	0.0003
(x,y,θ)	1.1×10^{-5}	1.1×10^{-5}	9.1×10^{-6}	8.3×10^{-6}	8.5×10^{-6}	1×10^{-5}
	Training data: Rotational motion					
Rotational	4.0×10^{-6}	2.9×10^{-6}	1.9×10^{-6}	4.0×10^{-6}	3.1×10^{-6}	4.2×10^{-6}
motion	2.9×10^{-6}	3.9×10^{-6}	3.8×10^{-6}	4.1×10^{-6}	2.5×10^{-6}	2.6×10^{-6}
(x,y,θ)	0.00117	0.00128	0.00128	0.00121	0.00128	0.00124
	Training data: Stationary motion					
Stationary	6.5×10^{-5}	6.7×10^{-5}	7.7×10^{-5}	7.7×10^{-5}	6.5×10^{-5}	7.9×10^{-5}
motion	4.3×10^{-5}	6.7×10^{-5}	7.3×10^{-5}	5.5×10^{-5}	7.1×10^{-5}	6.4×10^{-5}
(x,y,θ)	8.1×10^{-5}	9.8×10^{-5}	9.9×10^{-5}	8.4×10^{-5}	8.8×10^{-5}	0.00011

Even though adding control without memory did not improve the predictions, the NNs that contained memory as well as control inputs did outperform the NNs with memory. However, NNs containing more than three previous states did not improve the accuracy of the NNs, with most reaching similar errors to NNs with a memory of three. The one exception observed was when a memory of seven was used, where the NN's accuracy decreased. The cause of the decrease may have been due to the NNs failing to converge to the same local minimum. Nevertheless, the estimates with a memory of seven still outperformed the NNs that did not contain any control inputs.

Another notable observation was that adding control with memory resulted in decreases for both the arc and linear motion, even though the models were only trained with arc motion. In comparison, adding control had almost no effect on stationary motion, while only improving the yaw estimates

Table 5.10. The overall AE_{median} error (measured in meters and radians) for a NN with 11 hidden nodes trained with the arc motion simulated datasets using various amounts of memory.

	Memory (No Control)			**Memory (Control)**		
	1	**7**	**11**	**1**	**7**	**11**
Test Data	**Training data: Arc motion**					
Arc motion (x,y,θ)	0.057 0.041 0.016	0.254 0.280 0.089	0.422 0.464 0.149	0.085 0.036 0.0016	0.0082 0.0372 0.0018	0.0018 0.0012 0.0014
Linear motion (x,y,θ)	0.071 0.041 0.009	0.339 0.379 0.0006	0.563 0.634 0.0009	0.096 0.043 0.0016	0.0054 0.0016 0.00015	0.0021 0.0016 7×10^{-5}
Rotational motion (x,y,θ)	0.025 0.020 0.006	0.0017 0.0012 0.089	0.00082 0.00065 0.1496	0.036 0.039 0.001	0.0023 0.0019 0.0019	0.0009 0.0006 0.0017
Stationary motion (x,y,θ)	0.041 0.031 0.0036	0.0017 0.0009 0.0006	0.0010 0.0006 0.001	0.035 0.037 0.0008	0.0025 0.0024 0.0001	0.0008 0.0005 3.1×10^{-5}

for rotational motion. Thus including control generally improves the NNs predictions, even though a certain type of motion was not specifically used during training.

5.4.2.4 Training data spectrum tests

The subsequent tests investigated whether an increase in training data improved the estimates of the NNs. Both implementations with and without control were tested with the increase in training data. The subsequent tests focused on increasing the arc motion's training data, which performed worst in the previous tests as well as being the prevalent motion for a differential drive system.

The number of initial poses used to generate the training sets was increased from 16 to 29, 57 and 113 poses respectively. However, no improvement was observed for the NNs, as shown in Table 5.11. In particular, the results for NNs trained without control remained almost identical regardless of the

Table 5.11. The overall AE_{median} error (measured in meters and radians) for a NN with 11 hidden nodes using different numbers of initial poses.

	Initial poses			
	16 (original)	**29**	**57**	**113**
Test Data	**Training data: Arc motion**			
Arc	0.085	0.085	0.085	0.085
motion	0.095	0.094	0.095	0.095
(x,y,θ)	0.029	0.029	0.029	0.029
Linear	0.114	0.113	0.113	0.114
motion	0.126	0.126	0.127	0.127
(x,y,θ)	0.0002	0.0002	0.0002	0.0002
Rotational	0.0016	0.0008	0.0011	0.0016
motion	0.0008	0.0009	0.0011	0.0005
(x,y,θ)	0.0299	0.0299	0.030	0.030
Stationary	0.0011	0.0007	0.0011	0.0016
motion	0.0006	0.0007	0.0006	0.0007
(x,y,θ)	0.0006	0.0002	0.0002	0.0002
	Training data: Arc motion with control			
Arc	0.0019	0.0019	0.0021	0.0019
motion	0.0018	0.0015	0.002	0.0013
(x,y,θ)	0.0012	0.0012	0.0012	0.0012
Linear	0.0022	0.0018	0.0025	0.002
motion	0.0018	0.0016	0.0019	0.00132
(x,y,θ)	6.6×10^{-5}	5.1×10^{-5}	4.5×10^{-5}	4×10^{-5}
Rotational	0.0012	0.0009	0.0009	0.0008
motion	0.0014	0.0010	0.0013	0.0007
(x,y,θ)	0.0013	0.0013	0.0013	0.0013
Stationary	0.0008	0.0008	0.0008	0.0008
motion	0.0007	0.0009	0.0006	0.0005
(x,y,θ)	2.4×10^{5}	4.7×10^{-5}	2.2×10^{-5}	1.8×10^{-5}

number of initial poses, while the NNs with control showed a small variation at different numbers of training poses used.

Similarly, using additional controls during training did not offer marked improvements of the predictions. Table 5.12 shows that additional controls slightly reduces the accuracy of the estimates. Thus additional training data does not appear to increase the model's accuracy during training. As a last test, the number of initial poses was increased to 29, with 100 controls used at each initial pose. As with previous observations, the results were similar to those found previously with an error of $[0.0023, 0.003, 0.00118]$ for tested arc motion.

Table 5.12. The overall AE_{median} error (measured in meters and radians) for the NN with 11 hidden nodes using differing numbers of controls.

	Number of controls			
	10 (original)	**100**	**300**	**418**
Test Data	**Training data: Arc motion**			
Arc	0.085	0.0845	0.085	0.085
motion	0.095	0.096	0.095	0.095
(x,y,θ)	0.029	0.029	0.029	0.029
	Training data: Arc motion with control			
Arc	0.0019	0.0014	0.0019	0.0026
motion	0.0018	0.0024	0.0031	0.00195
(x,y,θ)	0.0012	0.0012	0.0012	0.0013

Furthermore, from Table 5.11 it is evident that the NNs were better able to represent stationary motion even though the NNs were trained using arc motion. In addition, the median deviations and skew for the stationary motion tests were significantly lower than in the arc motion tests. The implication is that the NNs were either learning to adjust the current state by very small amounts or to reproduce the current state.

5.5 SUMMARY

A summary of the experimental results were provided in this chapter. The three main experiments evaluated the performance of different NN configurations, using both tanh and linear activation

functions. The results indicated that tanh activation functions lead to warping during predictions. In addition, changes to the data's format revealed that splitting the datasets at discontinuities resulted in better convergence of the NNs. However, the SLAM algorithms with data association taken into consideration were still unable to provide accurate predictions.

Consequently, the chapter evaluated the performance of the learned models using the model learning metrics defined. For model learning evaluation both real-world and simulated datasets were trained and evaluated, as well as the effects of including control during training. The metrics revealed that increasing the amount of memory lead to a degradation in performance of the NNs. Adding control during training, in comparison, significantly improved the prediction accuracy of the NNs. However, diversifying the training data lead to almost no improvements in all test cases.

The NNs trained with the special-case motions also had significantly smaller errors than arc motion. Moreover, the arc motion training sets provided the best estimates when tested with stationary motion. Using the aforementioned observations, a conclusion was reached that NNs were not learning any of the higher-order dynamics or kinematics.

CHAPTER 6 DISCUSSION

6.1 CHAPTER OVERVIEW

The primary focus of the research was to learn motion models by making use of low-order state data and to employ the learned models in a SLAM context. To facilitate learning, a number of facets of the data and learning methodology required investigation. Section 6.2 provides a concise overview of the experimental results obtained in order to address the research questions.

In addition, Section 6.3 briefly investigates alternative machine learning strategies that could potentially be used to learn a vehicle's motion model. In particular, alternate recurrent NN implementations, particle swarm optimisation (PSO) [111, 112] and genetic programming [113] are investigated. Each solution was evaluated using the model learning strategy for straightforward comparison to the previous results.

6.2 LEARNING DATA-DERIVED NON-ANALYTICAL MODELS

6.2.1 Data formatting

The results obtained from both the Freiburg and simulated datasets demonstrated that pose data need to be carefully handled. Determining the sampling rate is an obvious filtering method to use when using data with a high frame-rate. Sub-sampling is also one of the most important, as the sampling rate dictates the rate at which the learned model can produce predictions. However, by sub-sampling the data, the amount of usable training data was drastically reduced. Decimation allowed the subtle

differences that were discarded by sub-sampling to be captured and included in the training data. As a consequence, the diversity for a single trajectory could be increased.

While sub-sampling and decimation made alterations to the datasets, the information contained was the same as the original set. Splitting the datasets at the yaw discontinuities, in comparison, changed the datasets into smaller subsets. By removing the discontinuities from the learning process, the NNs were able to converge to significantly more accurate results, as discussed in Section 5.2.2.3. However, the yaw discontinuities needed to be handled by the SLAM algorithm.

Iteratively appending datasets was found to have a small effect on the entire learning process. While the strategy did reduce the computational time required to reach a solution, almost no changes were observed on the training error, except in rare cases. An application where iteratively appending datasets might be useful is during online learning, where a "basic" model is learned with additional pose information added to refine the NNs as the vehicle traverses an environment. Scaling the training data also has a negative impact on the entire system. While not in itself the cause, combined with a non-linear activation function, scaling can warp the predictions. Hence any non-linear activation functions and scaling should not be applied to the datasets.

A further difficulty observed with real-world datasets was that only a limited amount of any type of motion was observable at specific poses. Hence the creation of simulated datasets aimed to overcome the limited training data by decomposing specific motion into a number of trajectories. While the simulated datasets cannot reproduce conditions in the real world, the simulated data does offer a simpler framework to test the learning approach. Specifically, the simulated datasets can be used to generate specific types of motion that can be evaluated in isolation, thus leading to additional knowledge of a learning algorithm's performance.

Once a methodology has been shown to learn the actual motion model, one can revert to using real-world datasets. While the Freiburg datasets offered a large variety of motion, not all types of motion were documented. Creating real-world datasets that contain extensive examples of different types of motion would therefore benefit learning algorithms such as these. Specifically, additional information on the vehicle state should be included that provides the wheel torque, friction coefficients of the surface and detected wheel-spin or skid, which would allow for a more in-depth evaluation of the

motion type. In addition, a large number of vehicle control and initial poses will need to be generated to cover the problem space, similar to what was done with the simulated datasets.

6.2.2 Model learning metrics

The experimental results demonstrated that training error metrics were insufficient to establish if a motion model had been learned. Analysis of the NNs' weights and the weight changes over training epochs could highlight some of the discrepancies. In particular, the significant contribution of the bias variable to the prediction indicates that a model could not be learned. Similarly, if large changes in the weights are observed during the latter stages of training with no marked improvement on the error, the changes could indicate that an incorrect solution has been reached.

While investigation of a network's weights can provide an indication of discrepancies during training, the weights cannot be used to measure if a correct model had been learned. Furthermore, the weights could not be used to determine the nature of the models that were learned, which led to the creation of the strategy for measuring the types of motion learned. The resulting measurements revealed that different types of motion had an impact on the network's prediction accuracy at all levels of observation. In addition, testing the models at the trajectory and control level revealed that the error increased as the control velocities increased.

Comparisons of the different metrics used during evaluation also revealed that some metrics were too optimistic, while others could be biased by outliers. In particular, yaw prediction errors were significantly influenced by discontinuities when measured with the MAE, MSE, standard deviation and variance. In comparison, the outliers' influence was limited when using the AE_{median} and $AE_{medianAD}$ metrics.

Large skew and kurtosis values were also regularly observed during the model learning tests. While some of the large kurtosis values can be explained by the outliers caused by the discontinuities, the values still indicate that some bias is present in the learned models. Hence both should be used with the AE_{median} and $AE_{medianAD}$ metrics to determine the overall performance of the model over the problem space.

6.2.3 The impact of memory

The primary motivation for using memory during the learning process was to allow the learning algorithm to encapsulate the higher-order dynamics of the system. Consequently, a number of experiments aimed to find the optimal amount of memory required by the system. Initial tests conducted showed that an increase in memory did not lead to a significant increase in errors except at the discontinuities. In comparison, the evaluation methodology used to determine if vehicle motion models could be learned (Section 5.3) indicated that an increase in memory led to significantly worse estimates. Furthermore, the NNs with no memory were observed to have the least error (when no control was included) using the evaluation methodology.

Some of the degradation of including additional memory can be explained by the shift-register containing the discontinuities for longer periods of time. However, the discontinuities in the test data were irregular, and can therefore not account for all of the performance degradation. In particular, the linear motion tests were observed to have larger errors with an increase in memory, even though no discontinuities were present in the test trajectories.

Conversely, when control was included during the learning process the NNs with no memory performed worst. Moreover, the number of previous states in the shift register did not have a large impact when control was included, with most of the NNs yielding similar errors. A likely explanation of the observed results is that the NNs are not learning the higher-order dynamics from the memory. Instead, the NNs might only be learning some delta of the previous states or learning to ignore most of the memory when control is included.

6.2.4 Learning the higher-order dynamics

The learning approach followed demonstrated that models could not be learned with sufficient accuracy. The results revealed that the NNs only provide some degree of accuracy when control was included as input (Section 5.4.2.2). Specifically, the models learned using the simulated linear, rotational and stationary motion seem to be able to provide fairly accurate estimates. As noted previously, all these motions were considered "special-case", as all the state variables do not interact with one another during execution.

For linear motion the yaw estimates remain stationary, with the x and y locations dependent on the previous x, y pair. While the x and y states do interact with each other according to the differential drive's motion model (see Section 3.2.2), the interaction is limited because the rotational velocity is zero (or at least very close to zero). Thus the motion can easily be approximated by a delta calculation or by using the forward velocity. Similarly, the rotational motion's yaw is the only variable that changes with time and can therefore either be calculated by using the angular velocity or through a delta of the previous yaw values.

Tests conducted using arc motion as training data produced the smallest errors when tested with stationary motion (see Table 5.10 and Table 5.11). Further examination of the results revealed that the larger the forward and angular velocities tested for arc motion, the larger the prediction error. Using linear and rotational motion as training and test data also resulted in larger prediction errors as the velocities increased. Hence, the prediction error scaled with the increase in velocity. Similar observations were noted when control was added to the training data. The only discernible difference between the NNs trained with and without control was that the prediction errors were less pronounced when using control.

The expectation was that the NNs would be able to encapsulate the higher-order dynamics from historic data in its structure. In addition, the NNs should be able to predict the vehicle's next state, given that the higher-order dynamics were learned. However, given the aforementioned observations, the NNs could, at best, learn some form of delta using the previous states and control variables. At worst, the models were only able to learn to reproduce the current state from the state input and memory. Two factors are largely responsible for the NNs failure to learn vehicle motion: The NNs ability to model complex functions and the input data format supplied to the NNs.

While NNs are able to model any mathematical function [114], learning complex mathematical functions, where the variables interact with additions, multiplications and periodic functions, seem to be unachievable using NNs. While not explicitly shown, tests conducted on NNs with a mixture of activation functions within each layer were also unable to overcome the deficiencies in training. Furthermore, NNs aim to learn the least complex function that fits the data in order to avoid over-fitting (through regularisation). Hence the fact that previous states are included as inputs suggests that the NNs will either learn a delta to add to the current state or simply reproduce the current vehicle state. Consequently, the higher-order dynamics for an N-dimensional motion model cannot be learned with a

TDL-NN that only uses low-order state data with memory.

6.2.5 Data-derived non-analytical models in a SLAM context

While the simulated and real-world SLAM experiments that were conducted did not yield any noteworthy results, the experiments did demonstrate that non-analytical models can be used in a SLAM context. The simulated SLAM experiments (Section 5.3.2.2) demonstrated that if the full map and motion state could be observed, the estimates were fairly close to the ground-truth. The results, however, were mainly due to the measurement model being able to observe the full vehicle state with minor contributions from the actual motion model.

The only observable influence that the transitional model had on the simulated dataset was the oscillations that occurred during phase changes. Tests conducted on the amount of assumed noise for both the transitional and measurement models showed that oscillations occur with more frequency as more belief is placed in the transitional model. Removing compensation for the orientation's in the EKF seemed to limit the oscillations significantly. However, even with the removal, the system still could not provide accurate results when data association was taken into account. Consequently, the suitability of using a data-derived model in an EKF-SLAM algorithm cannot be substantiated at this stage, as proof of a working model first needs to be obtained.

6.3 ALTERNATIVE LEARNING APPROACHES

The following section describes some of the alternative machine learning methodologies that were briefly tested to determine if a vehicle's motion model could be learned. In particular, tests were conducted using other recurrent structures, particle swarm optimisation [111] and genetic programming [113]. For the tests, the original simulated arc motion training sets were used, containing 15 initial poses with 10 different controls.

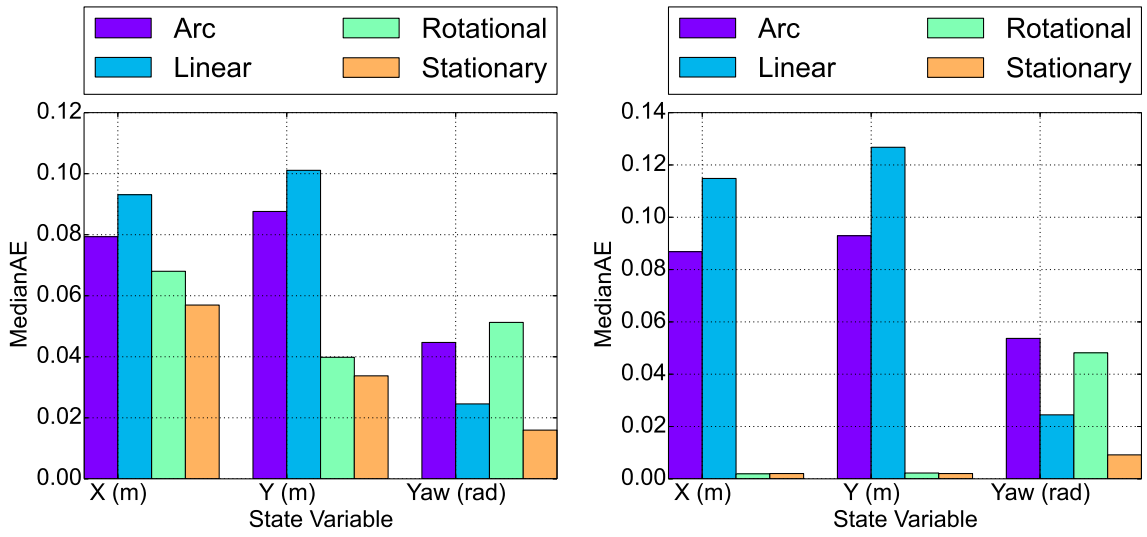
6.3.1 Other recurrent neural network structures

Alternative NNs that are regularly used with time-series data are recurrent neural nets (RNN) [115, 116] and long short-term memory (LSTM) neural nets [117, 118, 119]. The primary difference between

RNNs and tapped delay-line networks is that memory is modelled as the layer's output weights instead of the previous states. The weights are consequently fed back into the layer as additional inputs in order to model the memory. LSTM networks, in comparison, contain a separate memory block with gated activation units to select the memory and inputs to use for the predictions. Typically, LSTMs also contain "forget" gates that reset the memory blocks once the information becomes outdated.

Both implementations were briefly evaluated to determine if notable improvements could be observed during evaluation. The standard RNN and LSTM implementations available from Keras [120] were used with linear activation functions. A single hidden-layer network with 15 hidden nodes with a state memory of three was used to train the NNs. Furthermore, the MSE metric was used during training to ascertain each network's fitness.

In Figure 6.1 the AE_{median} summary for both the RNN and LSTM networks are provided. Both the LSTM and RNN networks produced similar errors to the TDL-NNs when tested with arc motion. However, when tested with the other types of motion, both the RNN and LSTM networks performed worse. Specifically, the RNN had larger yaw errors while the LSTM networks' estimates were worse across all three state variables. Preliminary results therefore indicate that neither of the alternative recurrent NN implementations offered direct improvements during motion model learning.



(a) LSTM. **(b)** RNN.

Figure 6.1. Summary of the AE_{median} for the alternative NNs trained with Keras.

6.3.2 Particle swarm optimisation

Particle swarm optimisation [111, 112] makes use of swarm theory to search for an optimal solution within a problem space. PSO implementations rely on a population, or swarm, of individual particles to emulate the social behaviour of bird flocking. A swarm will keep track of the location of the hyper-parameters with the least error, as well as each individual particle's best performing error to calculate the velocity of each particle. A random acceleration constant and velocity multiplier are also commonly added to ensure that the particles overshoot the current best solutions. Hence each particle will strive to reach the best solution through an indirect trajectory. Consequently the particles search through the problem space, covering most of the possible solutions.

As PSO is a search algorithm, any arbitrary function can be used as long as the hyper-parameters are defined. Given that the NNs failed to learn the motion's mathematical function, PSO will make use of an alternative solution to represent an arbitrarily complex function. To achieve such a function, a "dictionary" of base functions were defined in terms of the inputs (**x**) and hyper-parameters (**w**). The base functions selected were constant, scaling, multiplication, division and periodic functions, as shown below:

$$y_{constant} = \sum_{i=0}^{i=n} w_i \tag{6.1}$$

$$y_{scale} = \sum_{i=0}^{i=n} w_i x_i \tag{6.2}$$

$$y_{sin} = \sum_{i=0}^{i=n} w_i \sin(x_i) \tag{6.3}$$

$$y_{cos} = \sum_{i=0}^{i=n} w_i \cos(x_i) \tag{6.4}$$

$$y_{mutiply} = \sum_{i=0}^{i=n} \sum_{j=0}^{j=n} w_{(jn+i)} x_i x_j \tag{6.5}$$

$$y_{divide} = \sum_{i=0}^{i=n} \sum_{j=0}^{j=n} w_{(jn+i)} \frac{x_i}{x_j}. \tag{6.6}$$

Combining all the base functions (as shown in (6.7)) leads to a full representation that should be able to represent a large range of mathematical models. The one caveat of using a number of base functions is that each input and output variable increases the number of hyper parameters by $m \times (2n^2 + 4n)$,

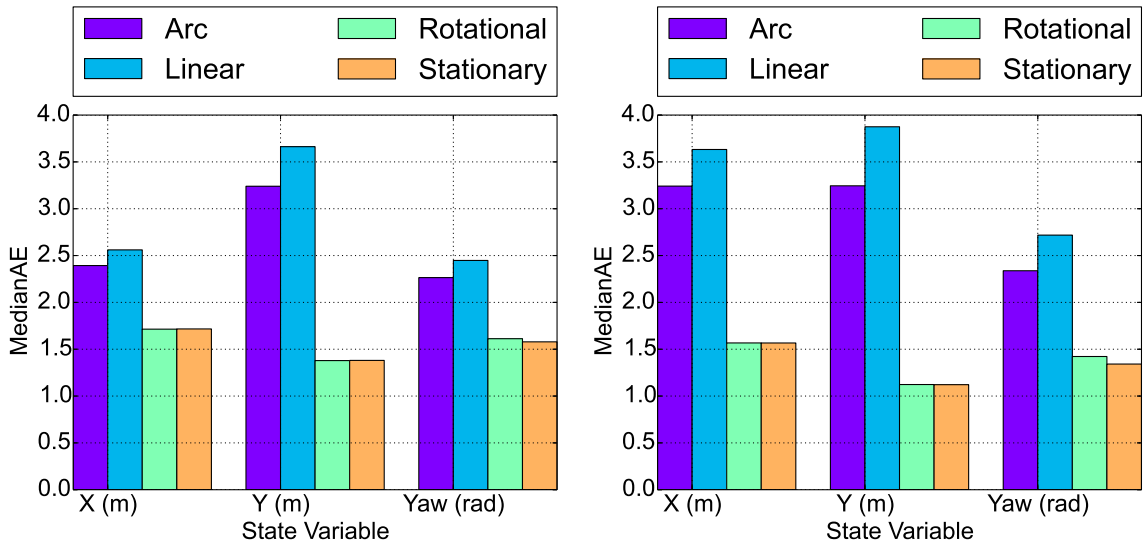
where n is the number of inputs and m is the number of outputs. Assuming a memory of 3, the number of hyper-parameters will therefore be 720. As such, one would expect most of the hyper-parameters to be sparse, with only a few contributing to the overall function. The concept is similar to dictionary learning approaches [121], with the exception of sparsely representing the signal. Instead, a set of functions is learned to represent an output that is not necessarily the same length as the input.

$$y_{est} = y_{constant} + y_{scale} + y_{sin} + y_{cos} + y_{mutiply} + y_{divide}. \tag{6.7}$$

The PSO implementations were evaluated to determine if the strategy could learn the motion model. The impact of the amount of memory was evaluated by training the PSO implementations with one and three previous states, as well as the impact of control on the results. For training, the swarm size was set to 300 particles, which is approximately 40% of the total number of hyper-parameters when using a memory of 3. Thus the initial particle initialisation should cover a relatively diverse parameter space. The hyper-parameters were randomly initialised between the interval $[-2.0, 2.0]$. The assumption was that constant and scaling factors would have a limited contribution to the full solution, with the hyper-parameters mainly used to "activate" a particular function.

By convention the maximum particle velocity is set to the absolute value of the initialisation interval, while an acceleration constant of 2.0 is selected . For the tests the maximum particle velocity was set to 4.0 to allow each particle to traverse the expected problem space. Furthermore, the maximum number of training epochs was set to 2000 as the expectation was that each particle would reach relatively stable solutions by that time. Lastly, the *MSE* metric was used to establish the fitness of each particle in the swarm during training.

Observations of the results (Figure 6.2) show that the PSO implementations performed significantly worse than the NNs (see Table 5.9 and Table 5.10 for a comparison). Further investigation revealed that most of the hyper-parameters at the end of training were non-zero. Thus each particle made use of the full function defined in (6.7) to predict the next state. The results therefore suggest that the PSO implementation cannot easily learn sparse representations of such an arbitrary function. Instead, the particles seem to converge on local minimums that minimise the function rather than learning the actual motion. Consequently, algorithms that are better able to learn sparse representations such as genetic programming were evaluated.



(a) PSO with no memory and no control.

(b) PSO with control and no memory.

Figure 6.2. Summary of the AE_{median} for the PSO implementation.

6.3.3 Genetic programming

Genetic programming refers to processes that learn a "program" or expressions using evolutionary techniques [113]. Typically genetic programs are built from a number of base functions with the terminals described through S-expressions (or LISP expressions). The full program can then be described using a tree structure containing an arbitrary number of nodes. As with genetic algorithms [122], genetic programming is based on a population that evolves through generations to search the problem space. Commonly the program's evolution is achieved through crossover, mutation, substitution, pruning and reproduction [123].

Hence the fittest individuals in each generation undergo modifications by combining sub-trees of individuals, randomly changing or removing sub-trees or simply copying the individual to the next generation. Various strategies have been proposed that use the semantics (base functions) to determine how individuals will be combined, many of which are briefly described in [123]. Commonly, the initialisation of genetic programs are implemented using the ramped "half-and-half" approach [113] that creates half of the individuals up to a specified depth (called the "full" method), while the rest are initialised with varying depths (called the "grow" method). Thus the "half-and-half" approach allows a population to represent various program structures, ensuring initial program diversity.

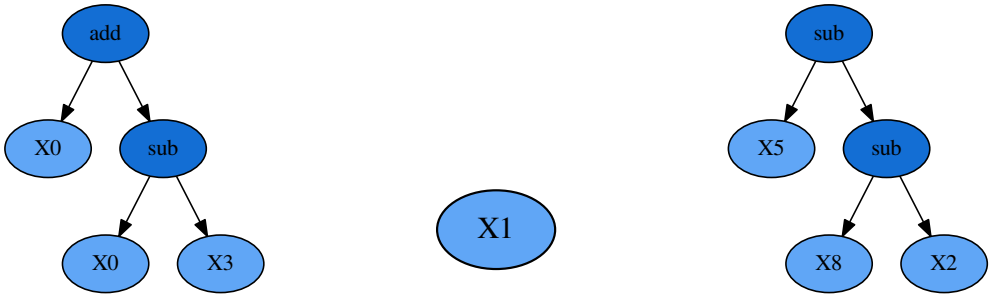
While genetic programs with multiple outputs have been used for both regression and classification [124], most implementations make use of only one output. The *gplearn* library [125] is such an implementation that allows for fast prototyping and testing, with the standard genetic programming measures implemented with one output per program. Consequently, *gplearn* was used to train vehicle motion models using the simulated datasets with the general training parameters described in Table 6.1.

Table 6.1. Parameters for the initial genetic program tests.

Parameter	**Value**	**Parameter**	**Value**
Crossover	[0.65, 0.4]	Hoist Mutation	[0.05, 0.1]
Point Mutation	[0.15, 0.2]	Subtree Mutation	[0.05, 0.25]
Initialisation Method	half-and-half	Reproduction	[0.1, 0.05]
Initialisation Depth	[5-10, 10-20]	Maximum Generations	2000
Base Functions	[addition, subtraction, multiplication, division, sin, cos]	Population Size	400

Training sets with and without control were also included in the evaluation using memory of either one or three states. Thus three separate programs were learned to predict each state variable, each using all the input data available. After training, the *gplearn* libraries were used to visualise the tree's structure, as illustrated in Figure 6.3 and Figure 6.4. In addition, the model learning metrics were used with the programs for a direct comparison to the previous strategies. Table 6.2 provides a summary of the AE_{median} error when the genetic programs were trained and tested with arc motion.

Observations of the model learning metrics combined with the graphed program structure showed that the genetic programs tended to learn a delta between the states when more than one previous state was used. Conversely, in certain cases the genetic program could only learn to reproduce the current state variable as the next state, as shown in Figure 6.3 (b). Furthermore, in the cases where a delta between the states was learned, the errors made by genetic programs were found to be smaller than the errors made by the NNs, which indicated that the NNs did not learn an actual delta of the states. Including the control variables with additional memory also did not appear to improve the estimates, with the genetic programs learning to ignore the control inputs.

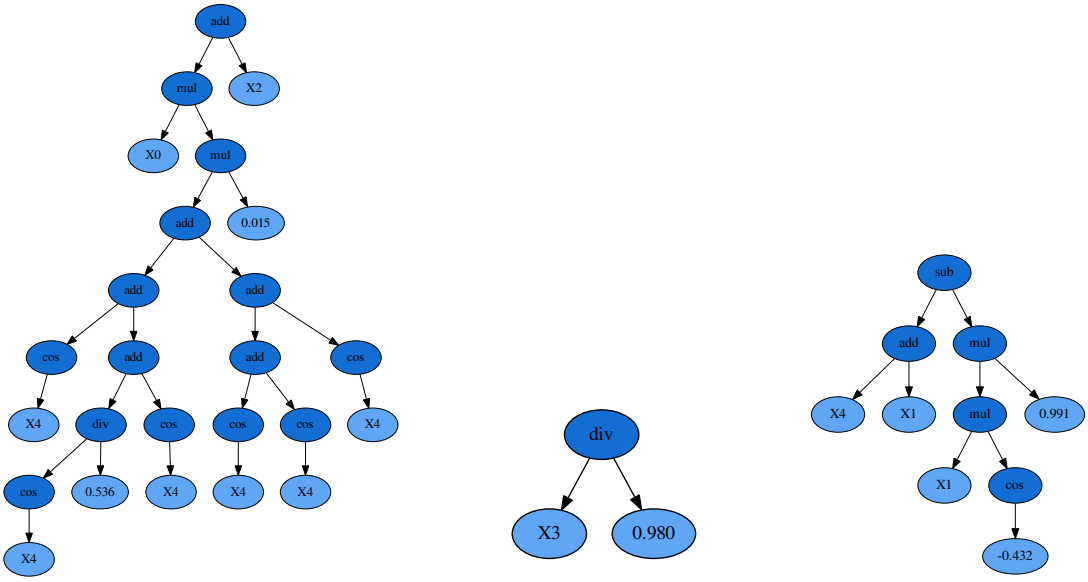


(a) Genetic program to predict $x(m)$.

(b) Genetic program to predict $y(m)$.

(c) Genetic program to predict $\theta(rad)$.

Figure 6.3. Genetic programs learned for each state variable using a memory of three without control and an initialisation depth of 5-10.



(a) Genetic program to predict $x(m)$.

(b) Genetic program to predict $y(m)$.

(c) Genetic program to predict $\theta(rad)$.

Figure 6.4. Genetic programs learned for each state variable using no memory with control inputs and an initialisation depth of 10-20.

Table 6.2. The overall AE_{median} error (measured in meters and radians) for the genetic programs trained with the simulated arc motion datasets.

Test Data	**Memory (No Control)**		**Memory (Control)**	
	1	**3**	**1**	**3**
	Initialisation depth of 5-10 and crossover of 0.65			
Arc motion (x,y,θ)	0.03512 0.046576 0.01499	0.00021 0.046157 0.01499	0.03318 0.03372 0.01499	0.04047 0.04615 0.01499
	Initialisation depth of 10-20 and crossover of 0.4			
Arc motion (x,y,θ)	0.0279 0.0392 0.0142	0.0405 0.0004 0.01499	0.0009 0.0372 5×10^{-5}	0.0405 0.0004 0.01499

The genetic programs trained with no memory, in comparison, were prone to learn that the next state was either the current state divided by some constant or the current state (see Figure 6.4 (b)). The only exception for the aforementioned results was observed when control was included; the initialisation depth increased and crossover percentage lowered. In this case the complexity of the programs learned increased significantly (see Figure 6.4 (a) and (c)). Furthermore, the AE_{median} for programs using a larger initialisation depth decreased, leading to more accurate predictions.

However, while these programs were able to learn more complex functions, the programs were still only dependent on the previous state, control variables and constants. The implication is that, at best, some of the vehicle's kinematics can be learned for a single state variable. As an example, consider the program that learned to predict the yaw (Figure 6.4 (c)). Substituting $X0$ - $X5$ for the corresponding input symbols and simplifying leads to (6.8), where the predicted yaw depends on the current yaw, angular velocity and some constant.

$$\begin{aligned} \theta_{k+1} &= (\theta_k + \omega) - (0.991\omega\cos(-0.432)) \\ &= \theta_k + \omega - 0.899\omega \\ &= \theta_k + 0.11\omega. \end{aligned} \tag{6.8}$$

Similarly, the predictions for the x state variable can be simplified to (6.9). While the equation does not contain all the kinematic or dynamic interactions, it does illustrate that more complex functions can be learned.

$$x_{k+1} = x_k + 0.083 v_f \cos(\theta_k). \tag{6.9}$$

The observations of the genetic programs trained therefore illustrate that providing states as memory will result, at best, in learning a delta of the states. Furthermore, including control variables as input during training were shown to result in learning at least part of the kinematics in a vehicle's motion. Thus none of the dynamics of a vehicle can be learned without significant modification to the input data format and evolutionary strategies currently used.

6.4 SUMMARY

The research questions were addressed in the first part of the chapter. The current input format of the data was found to be one of the primary short-comings during training. Another difficulty that the NNs faced was learning the complex mathematical operations required to encompass the vehicle's motion dynamics. Furthermore, the real-world datasets were found to be insufficient to train distinct motion types. Thus, simulated datasets were created to train a particular motion type while simultaneously simplifying the motion to learn.

Analysis of the evaluation metrics revealed that the standard training metrics could highlight obvious deficiencies during training. However, the standard metrics were largely insufficient to ascertain whether a model learned the actual motion. In comparison, the methodology defined to evaluate learned models could demonstrate differences between the NNs. In particular, the metrics showed a distinct performance difference when additional memory was used or control added and that preference should be given to metrics that are robust to outliers. Furthermore, it was concluded that TDL-NNs could not be used to learn the higher-order dynamics of a vehicle's motion. At best, the NNs could learn some delta between the states. At worst, the NNs were only learning to reproduce the state.

Incorporating the learned models into a SLAM system was also addressed. While the simulated landmarks tests could provide relatively accurate estimates, the yaw regularly oscillated once a

discontinuity was observed. In addition, the full SLAM implementation with data association could not provide accurate or stable predictions. Thus, incorporating learned models in an EKF-SLAM system could not be verified as a model that actually learns the motion first needs to be demonstrated.

Lastly, the chapter briefly investigated alternative machine learning approaches. Similar model learning results were obtained with RNNs and LSTMs, with the LSTMs producing worse estimates when tested with motion types that were not trained on. In addition, the PSO implementation performed significantly worse than the NNs, while the genetic programs showed some improvements under certain conditions. However, the genetic programs were still prone to learn a delta calculation when memory was included. Furthermore, individual programs needed to be trained for each state variable. Consequently, additional investigation is required to fully investigate a genetic programming approach.

CHAPTER 7 CONCLUSION

7.1 OVERVIEW OF THE WORK

The research conducted aimed to learn an abstract representation of a vehicle's motion model without knowledge of any of the higher-order dynamics and to incorporate the model in a SLAM system. To this end, the methodology made use of prior knowledge of the vehicle' state to learn a data-derived non-analytical model of the motion. The vehicle's state was limited to 2D motion that contained three DOF to describe the entire model. Models in the form of a generalised odometry model and a kinematic model for a differential drive were used as a baseline for comparison.

To facilitate learning a TDL-NN was implemented that encapsulated the memory of the network's input in a shift-register. The TDL-NN used two methodologies for the input-data shift-register: A straightforward shift register containing N previous states and a shift-register with bias inputs in the form of vehicle controls. The primary difference between the two approaches was that control, in the form of forward velocity and angular velocity, was added as additional input but not included in the shift-register.

Including the NNs in an EKF-SLAM implementation was subsequently investigated. As an EKF is only dependent on the previous state, the methodology for including an N^{th}-order Markov model needed to be defined. To reach a tractable solution the NNs were regarded as black-box estimators. Consequently, the EKF had no knowledge of the function that the NN had learned. As a result, the EKF's prediction step was handled by the NN at the cost of the effectiveness of the predictor's derivatives (Jacobian matrices). The rest of the EKF-SLAM algorithm then followed the general procedure to update the predictions using a measurement model of the environment.

Training the TDL-NN was implemented using the back-propagation algorithm. Extensive permutations of the parameters were tested to find the optimal configurations for the NNs. The type of activation function, learning rate and annealing factors were considered the most important hyper-parameters for the NNs. In addition, changes to the input data format such as decimation, splitting the data at discontinuities and sub-sampling the data were evaluated during learning.

Simulated datasets using a kinematic model of a differential drive vehicle were generated in order to determine the types of motion that could be learned. The number of initial poses and controls used to generate the datasets were varied to establish whether the scope of training data had any effect on the learning process. An evaluation methodology was also defined to test each motion type. The metrics comprehensively analysed the one-forward prediction using various statistical measures to ascertain the model's suitability.

Lastly, RNNs, LSTMs, PSO and genetic programming approaches were briefly investigated. The RNN, LSTM approaches followed very similar methodologies to the TDL-NN, with the difference being that no shift-register was used to encapsulate the memory. The PSO and genetic programming methodology, in comparison, defined a set of base functions that could be used to create a larger system. For the PSO implementation, the base functions were "activated" using the hyper-parameters, while genetic programming randomly generated trees and evolved them over time.

7.2 RESEARCH FINDINGS

Inherently learning a vehicle's dynamics using low-order state data is clearly a complex problem that has yet to be solved. The research conducted illustrated a number of problems that need to be addressed before such an approach can become feasible. Foremost is that NNs are incapable of learning the higher-order dynamics. The complex mathematical operations required by the motion model and the format of the data are the primary limiting factors that prevent the NNs from learning a motion model.

Neither linear nor non-linear activation functions seem to be capable of inherently encapsulating a complex mixture of functions in which the inputs can interact with one another. Specifically, the NNs seem incapable of learning complex operations such as the multiplication of two input variables

combined with additions and periodic functions. This is primarily due to the fact that standard NNs are based on a sum of the inputs and weights. Because different measurement units (meters and radians) are used and multiple complex mathematical functions need to be learned, the NNs do not have the capability to represent the dynamics.

Furthermore, standard evaluation methods applied during training do not provide an accurate indication that the NNs have learned a correct model. Examining each neuron's weight over time can indicate whether a stable solution has been reached, as well as which input has the largest impact on the prediction. An example of where the NNs did not train properly was observed when significant weight was allocated to the bias variable, leading to incorrect estimates. However, while a network's weights can provide an indication of problems during training, this does not guarantee that a correct solution has been reached.

Determining if a vehicle's motion model has been learned requires that the problem be evaluated at different levels of detail. Specifically, the learned models need to be evaluated at the trajectory, control and motion type levels to provide a comprehensive overview of the models' behaviour. Observations found that the AE_{median} metric was the most suitable to evaluate each motion error, as outliers caused by the discontinuities did not have as large an impact. Similarly, the $AE_{medianAD}$ is the preferred metric when evaluating the deviations between estimates.

The amount of memory contained in the shift-register did not yield any improvements during estimation either. Instead, the predictions were less accurate when larger amounts of memory were used. In addition, using more diverse training data did not yield any observable improvements using the model learning metrics. As these results are counter-intuitive, the indication is that the input data's format is unsuitable for a NN methodology. In particular, regularisation during training forces the NNs to eliminate complex solutions given the observed data. Coupled with the format of the input data, motion model learning is unlikely to learn anything more than a delta between the states. Similar results from the genetic programs corroborate the findings for the NNs and imply that alternative learning strategies that do not over-simplify the models needs to be considered.

7.3 FUTURE WORK

A number of avenues of further investigation were revealed during the course of the work. Foremost among these is implementing alternative learning strategies to learn the motion models. Genetic programming, in particular, is the favoured approach to investigate. However, a number of modifications are required before a solution that is able to learn the dynamics can be reached, such as allowing multiple outputs to each program. In addition, a methodology will need to be developed that forces the genetic programs to reduce the impact of regularisation. Hence the complexity of higher-order dynamics should be favoured over delta calculations. To this end, the methodology and format to supply input data will need to be re-evaluated.

Methods to automatically force the learning algorithms to incorporate the dynamics should also be investigated. The simplest method to incorporate the dynamics would be to include integrators, differentiators and matrix multiplication in the learning algorithm's methodology. Lagrange's equation of motion could also be incorporated for similar reasons. With genetic programs, the first approach should be fairly straightforward to implement. For the second approach, additional investigation will be required.

Thirdly, there is a need for the creation of real-world datasets that emulate the simulated training and test data. The datasets need to be created at various initial poses, with different controls at each pose. Ideally, the datasets should also contain additional information on the dynamics of the platform, such as wheel torque, wheel slippage, center of mass and friction coefficients of the wheels and surface. The datasets should also define more diverse motion types for vehicle motion that include datasets where wheel slippage is prevalent, where backwards motion is possible and where acceleration is present. Lastly, different surface types can be included in the datasets to determine the impact of friction on a vehicle's motion.

Another avenue that requires investigation is incorporating the learned models into recursive Bayesian estimators. While the current strategy was tested during the course of the work, unambiguous proof that incorporating learned models into recursive Bayesian estimations improves estimates during localisation and mapping is still required. Furthermore, research on how to include N^{th}-order Markov models into recursive Bayesian estimators still needs to be conducted. Specifically, removing the

assumption that the learned model is a black-box estimator, and therefore independent, needs to be investigated and the resulting estimator's performance determined.

Further extensions of the recursive Bayesian estimators that may yield promising results are training multiple models for a vehicle and encapsulating them through an IMM filter. For example, models could be trained on specific motion types (such as arc, stationary, skidding or acceleration) or for different surfaces and then included in a larger system that statistically biases the predictions towards the most likely model. Lastly, the work still needs to be extended to other vehicle types. Drive types such as skid-steer models, Ackerman steering and omni-drive systems are examples of wheeled vehicles that operate on a 2D plane. The learning strategy should also be extended to vehicles operating on a 3D plane, such as quad-copters and legged robots.

7.4 SUMMARY

The work conducted revealed that NNs, while theoretically able to model any arbitrary function, are largely unsuitable for complex regression problems. In particular, low-order state memory cannot be used as a substitute to encapsulate the higher-order dynamics to provide predictions. Consequently, alternative approaches need to be investigated that can facilitate learning using low-order state data, such as genetic programs. In addition, there is a need to create real-world datasets that can be divided into distinct motion types and comprehensively cover the possible movements of a particular platform.

Incorporating N^{th}-order Markov processes and including them in a recursive Bayesian estimator is also required once a suitable model has been learned. Once a simple kinematic motion model can be learned using low-order state data and incorporated into a SLAM system, the work can be extended to more complex models. Specifically, vehicles with complex kinematic and dynamic configurations could then potentially be learned without any knowledge of the vehicle's characteristics.

REFERENCES

[1] R. Smith, M. Self, and P. Cheeseman, *Estimating Uncertain Spatial Relationships in Robotics*. New York, NY: Springer New York, 1990, pp. 167–193.

[2] H. Durrant-Whyte and T. Bailey, "Simultaneous Localization and Mapping: Part I," *Robotics & Automation Magazine*, vol. 13, pp. 99–108, June 2006.

[3] T. Bailey and H. Durrant-Whyte, "Simultaneous Localization and Mapping (SLAM): Part II," *IEEE Robotics & Automation*, vol. 13, pp. 108–117, September 2006.

[4] M. Dissanayake, P. Newman, S. Clark, H. Durrant-Whyte, and M. Csorba, "A Solution to the Simultaneous Localization and Map Building (SLAM) Problem," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 3, pp. 229–241, June 2001.

[5] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, 1st ed. Cambridge, MA: MIT Press, 2006.

[6] M. Cummins and P. Newman, "FAB-MAP: Probabilistic Localization and Mapping in the Space of Appearance," *The International Journal of Robotics Research*, vol. 27, no. 6, pp. 647–665, June 2008.

[7] J. Civera, O. G. Grasa, A. J. Davison, and J. M. M. Montiel, "1-point RANSAC for EKF-based Structure from Motion," in *IEEE International Conference on Intelligent Robots and Systems*, December 2007, pp. 3498–3504.

[8] G. Grisetti, D. Lodi Rizzini, C. Stachniss, E. Olson, and W. Burgard, "Online Constraint Network Optimization for Efficient Maximum Likelihood Map Learning," in *IEEE International Conference on Robotics and Automation*, May 2008, pp. 1880–1885.

[9] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o : A General Framework for Graph Optimization," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, May 2011, pp. 3607–3613.

[10] J. Schwendner, S. Joyeux, and F. Kirchner, "Using Embodied Data for Localization and Mapping," *Journal of Field Robotics*, vol. 31, no. 2, pp. 263–295, 2014.

[11] A. Mandow, J. L. Martinez, J. Morales, J. L. Blanco, A. Garcia-Cerezo, and J. Gonzalez, "Experimental kinematics for wheeled skid-steer mobile robots," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2007, pp. 1222–1227.

[12] Z. Wang and G. Dissanayake, "Exploiting Vehicle Motion Information in Monocular SLAM," in *12th International Conference on Control Automation Robotics Vision (ICARCV)*, December 2012, pp. 1030–1035.

[13] A. Karakasoglu and M. K. Sundareshan, "A Recurrent Neural Network-based Adaptive Variable Structure Model-following Control of Robotic Manipulators," *Automatica*, vol. 31, no. 10, pp. 1495–1507, 1995.

[14] D. Vasquez, T. Fraichard, and C. Laugier, "Incremental Learning of Statistical Motion Patterns with Growing Hidden Markov Models," *IEEE Transactions on Intelligent Transportation Systems*, vol. 10, no. 3, pp. 403–416, 2009.

[15] J.-G. Kang, S. Kim, S.-Y. An, and S.-Y. Oh, "A New Approach to Simultaneous Localization and Map Building with Learning: NeoSLAM (Neuro-Evolutionary Optimizing)," *Applied Intelligence*, vol. 36, no. 1, pp. 242–269, 2012.

[16] B. T. Morris and M. M. Trivedi, "Learning, Modeling, and Classification of Vehicle Track Patterns from Live Video," *IEEE Transactions on Intelligent Transportation Systems*, vol. 9,

no. 3, pp. 425–437, 2008.

[17] L. Wang, N. Zhang, and H. Du, "Real-time Identification of Vehicle Motion-Modes Using Neural Networks," *Mechanical Systems and Signal Processing*, vol. 50-51, pp. 632–645, 2015.

[18] K. S. Hatamleh, M. Al-Shabi, A. Al-Ghasem, and A. A. Asad, "Unmanned Aerial Vehicles Parameter Estimation using Artificial Neural Networks and Iterative Bi-Section Shooting Method," *Applied Soft Computing*, vol. 36, pp. 457–467, 2015.

[19] J.-C. Sun, N. Wang, M. J. Er, and Y.-C. Liu, "Extreme Learning Control of Surface Vehicles with Unknown Dynamics and Disturbances," *Neurocomputing*, vol. 167, pp. 535–542, 2015.

[20] K. S. Narendra and K. Parthasarathy, "Identification and Control of Dynamical Systems Using Neural Networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4–27, 1990.

[21] M. Choi, R. Sakthivel, and W. K. Chung, "Neural Network-Aided Extended Kalman Filter for SLAM Problem," in *Proceedings of the 2007 IEEE International Conference on Robotics and Automation*, April 2007, pp. 1686–1690.

[22] N.-B. Hoang and H.-J. Kang, "Neural Network-based Adaptive Tracking Control of Mobile Robots in the Presence of Wheel Slip and External Disturbance Force," *Neurocomputing*, vol. 188, pp. 12–22, 2016.

[23] J. Ko, D. J. Klein, D. Fox, and D. Hähnel, "GP-UKF: Unscented Kalman filters with Gaussian process prediction and observation models," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2007, pp. 1901–1907.

[24] S. Thrun and M. Montemerlo, "The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures," *International Journal of Robotics Research*, vol. 25, no. 5, pp. 403–429, May 2006.

[25] E. Olson, J. Leonard, and S. Teller, "Fast Iterative Alignment of Pose Graphs with Poor Initial Estimates," in *Proceedings of the 2006 IEEE International Conference on Robotics and*

Automation, May 2006, pp. 2262–2269.

[26] G. Klein and D. Murray, "Parallel Tracking and Mapping for Small AR Workspaces," in *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, 2007, pp. 1–10.

[27] K. Konolige and M. Agrawal, "FrameSLAM: From bundle adjustment to real-time visual mapping," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1066–1077, October 2008.

[28] R. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Fluids Engineering*, vol. 82, pp. 35–45, 1960.

[29] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," in *Association for the Advancement of Artificial Intelligence (AAAI)*, 2002, pp. 593–598.

[30] ——, "FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges," in *Proceedings of the International Joint Conference on Artificial Intelligence*, 2003, pp. 1151–1156.

[31] F. Dellaert, S. Thrun, D. Fox, and W. Burgard, "Monte Carlo Localization for Mobile Robots," in *IEEE International Conference on Intelligent Robots and Systems Robotics & Automation*, no. May, 1999, pp. 1322–1328.

[32] J. S. Liu, R. Chen, and X. Xt, "Sequential Monte Carlo Methods for Dynamic Systems," *Journal of the American Statistical Association*, vol. 93, no. 443, pp. 1032–1044, September 1998.

[33] A. Doucet, S. Godsill, and C. Andrieu, "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering," *Statistics and Computing*, vol. 10, no. 2, pp. 197–208, August 2000.

[34] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: An efficient probabilistic 3D mapping framework based on octrees," *Autonomous Robots*, vol. 34, no. 3, pp. 189–206, February 2013.

[35] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A Benchmark for the Evaluation of RGB-D SLAM Systems," in *IEEE International Conference on Intelligent Robots and Systems*, 2012, pp. 573–580.

[36] S. Oh, M. Hahn, and J. Kim, "Dynamic EKF-based SLAM for Autonomous Mobile Convergence Platforms," *Multimedia Tools and Applications*, vol. 74, no. 16, pp. 6413–6430, 2015.

[37] X. Zhang, A. B. Rad, and Y.-K. Wong, "Sensor Fusion of Monocular Cameras and Laser Rangefinders for Line-Based Simultaneous Localization and Mapping (SLAM) Tasks in Autonomous Mobile Robots." *Sensors*, vol. 12, no. 1, pp. 429–52, January 2012.

[38] Y. Latif, C. Cadena, and J. Neira, "Robust Loop Closing over Time for Pose Graph SLAM," *The International Journal of Robotics Research*, vol. 32, no. 4, pp. 1611–1626, October 2013.

[39] W. Maddern, M. Milford, and G. Wyeth, "CAT-SLAM: Probabilistic localisation and mapping using a continuous appearance-based trajectory," *The International Journal of Robotics Research*, vol. 31, no. 4, pp. 429–451, April 2012.

[40] L. Pérez, L. L. M. Paz, J. Tardos, and J. Neira, "Divide and Conquer: EKF SLAM in $\mathcal{O}(n)$," *IEEE Transactions on Robotics*, vol. 24, no. 5, pp. 1107–1120, 2008.

[41] M. Cummins and P. Newman, "Appearance-only SLAM at Large Scale with FAB-MAP 2.0," *The International Journal of Robotics Research*, vol. 30, no. 9, pp. 1100–1123, November 2010.

[42] V. Indelman, S. Williams, M. Kaess, and F. Dellaert, "Information Fusion in Navigation Systems via Factor Graph Based Incremental Smoothing," *Robotics and Autonomous Systems*, vol. 61, no. 8, pp. 721–738, August 2013.

[43] R. C. Luo and C. C. Lai, "Multisensor Fusion-based Concurrent Environment Mapping and Moving Object Detection for Intelligent Service Robotics," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 8, pp. 4043–4051, August 2014.

[44] M. R. Perception and A. Elfes, "Using Occupancy Grids for Mobile Robot Perception and Navigation," *Computer*, vol. 22, no. 6, pp. 46–57, 1989.

[45] S. Thrun, "Learning Occupancy Grid Maps with Forward Sensor Models," *Autonomous Robots*, vol. 15, no. 2, pp. 111–127, 2003.

[46] R. Rusu and S. Cousins, "3D is Here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2011, pp. 1–4.

[47] S. Cruz, J. Wilhelms, and A. V. Gelder, "Octrees for Faster Isosurface Generation," *Association for Computing Machinery Transactions on Graphics (TOG)*, vol. 11, no. 5, pp. 57–62, 1992.

[48] M. Boucher, F. Ababsa, and M. Mallem, "Reducing the SLAM Drift Error Propagation Using Sparse but Accurate 3D Models for Augmented Reality Applications," in *Proceedings of the Virtual Reality International Conference: Laval Virtual*, New York, March 2013, pp. 1–6.

[49] B. Triggs, P. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle Adjustment - A Modern Synthesis," *Vision Algorithms: Theory and Practice*, vol. 34, pp. 298–372, 2000.

[50] G. Dubbelman and B. Browning, "Closed-form Online Pose-chain SLAM," in *International Conference on Robotics and Automation (ICRA)*, May 2013, pp. 5190–5197.

[51] J. Nilsson, J. Fredriksson, and A. C. E. Odblom, "Reliable Vehicle Pose Estimation Using Vision and a Single-track Model," *IEEE Transactions on Intelligent Transportation Systems Magazine*, vol. 15, no. 6, pp. 2630–2643, 2014.

[52] F. Solc and J. Sembera, "Kinetic Model of a Skid Steered Robot," in *Proceedings of the 7th WSEAS Conference on Signal Processing, Robotics and Automation*, February 2008, pp. 61–65.

[53] D. J. Montana and L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," *Proceedings of the 11th International Joint Conference on Artificial intelligence - Volume 1*, vol. 89, pp. 762–767, 1989.

[54] K. Jo, K. Chu, K. Lee, and M. Sunwoo, "Integration of Multiple Vehicle Models with an IMM Filter for Vehicle Localization," in *IEEE Intelligent Vehicles Symposium*, 2010, pp. 746–751.

[55] E. Mazor, A. Averbuch, Y. Bar-Shalom, and J. Dayan, "Interacting Multiple Model Methods in Target Tracking: A survey," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 34, no. 1, pp. 103–123, 1998.

[56] K. Jo, K. Chu, and M. Sunwoo, "Interacting Multiple Model Filter-based Sensor Fusion of GPS With In-vehicle Sensors for Real-time Vehicle Positioning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 1, pp. 329–343, 2012.

[57] Y. Tian, J. Zhang, and J. Tan, "Adaptive-Frame-Rate Monocular Vision and IMU Fusion for Robust Indoor Positioning," in *IEEE International Conference on Robotics and Automation (ICRA)*, May 2013, pp. 2257–2262.

[58] G. H. Lee, F. Fraundorfer, and M. Pollefeys, "Robust pose-graph loop-closures with expectation-maximization," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, November 2013, pp. 556–563.

[59] K. M. Brown and J. E. Dennis, "Derivative Free Analogues of the Levenberg-Marquardt and Gauss Algorithms for Nonlinear Least Squares Approximation," *Numerische Mathematik*, vol. 18, no. 4, pp. 289–297, 1971.

[60] R. A. Fisher, "The Use of Mutliple Measurements in Taxonomic Problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[61] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme Learning Machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1-3, pp. 489–501, 2006.

[62] K. Hornik, "Some New Results on Neural Network Approximation," *Neural Networks*, vol. 6, no. 8, pp. 1069–1072, January 1993.

[63] S. Z. Qin, H. T. Su, and T. J. McAvoy, "Comparison of Four Neural Net Learning Methods for Dynamic System Identification," *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 122–130, 1992.

[64] M. K. Sundareshan and C. Askew, "Neural Network-assisted Variable Structure Control Scheme for Control of a Flexible Manipulator Arm," *Automatica*, vol. 33, no. 9, pp. 1699–1710, 1997.

[65] M. Ertugrul and O. Kaynak, "Neuro Sliding Mode Control of Robotic Manipulators," *Mechatronics*, vol. 10, no. 1-2, pp. 239–263, 2000.

[66] G. L. Plett, "Adaptive Inverse Control of Linear and Nonlinear Systems Using Dynamic Neural Networks," *IEEE Transactions on Neural Networks*, vol. 14, no. 2, pp. 360–376, 2003.

[67] D. Vasquez, T. Fraichard, and C. Laugier, "Growing Hidden Markov Models: An Incremental Tool for Learning and Predicting Human and Vehicle Motion," *The International Journal of Robotics Research*, vol. 28, no. 11-12, pp. 1486–1506, 2009.

[68] A. Censi and R. M. Murray, "Learning Diffeomorphism Models of Robotic Sensorimotor Cascades," in *IEEE International Conference on Robotics and Automation*, no. 1, 2012, pp. 3657–3664.

[69] A. Censi, A. Nilsson, and R. M. Murray, "Motion Planning in Observations Space with Learned Diffeomorphism Models," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 2860–2867, May 2013.

[70] A. Censi and R. M. Murray, "Bootstrapping Bilinear Models of Simple Vehicles," *The International Journal of Robotics Research*, vol. 34, no. 8, pp. 1087–1113, 2015.

[71] R. Urtasun, D. J. Fleet, and P. Fua, "3D People Tracking with Gaussian Process Dynamical Models," in *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*, June 2006, pp. 238–245.

[72] J. M. Wang, D. J. Fleet, and A. Hertzmann, "Gaussian Process Dynamical Models for Human Motion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 2, pp. 283–298, 2008.

[73] N. D. Lawrence, "Gaussian Process Latent Variable Models for Visualisation of High Dimensional Data," in *Proceedings of the 16th International Conference on Neural Information Processing Systems*, 2003, pp. 329—-336.

[74] J. Ko and D. Fox, "GP-BayesFilters: Bayesian filtering using Gaussian process prediction and observation models," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, September 2008, pp. 3471–3476.

[75] A. Kemppainen, J. Haverinen, I. Vallivaara, and J. Roning, "Near-optimal SLAM Exploration in Gaussian Processes," *2010 IEEE Conference on Multisensor Fusion and Integration*, pp. 7–13, September 2010.

[76] I. Vallivaara, J. Haverinen, A. Kemppainen, and J. Roning, "Simultaneous Localization and Mapping Using Ambient Magnetic Field," in *2010 IEEE Conference on Multisensor Fusion and Integration*, September 2010, pp. 14–19.

[77] B. Ferris, D. Fox, and N. Lawrence, "WiFi-SLAM Using Gaussian Process Latent Variable Models," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, ser. IJCAI'07, 2007, pp. 2480–2485.

[78] J. Huang and D. Millman, "Efficient, Generalized Indoor WiFi GraphSLAM," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 1038–1043.

[79] C. H. Tong, P. Furgale, and T. D. Barfoot, "Gaussian Process Gauss-Newton for Non-parametric Simultaneous Localization and Mapping," *The International Journal of Robotics Research*, vol. 32, no. 5, pp. 507–525, 2013.

[80] S. C. Stubberud, R. N. Lobbia, and M. Owen, "An Adaptive Extended Kalman Filter using Artificial Neural Networks," in *Proceedings of the 34th IEEE Conference on Decision and*

Control, December 1995, pp. 1852–1856.

[81] J. G. Kang, S. Y. An, and S. Y. Oh, "Modified Neural Network Aided EKF based SLAM for Improving an Accuracy of the Feature Map," in *The 2010 International Joint Conference on Neural Networks*, July 2010, pp. 18–23.

[82] Q.-L. Li, Y. Song, and Z.-G. Hou, "Neural Network Based FastSLAM for Autonomous Robots in Unknown Environments," *Neurocomputing*, vol. 165, pp. 99–110, 2015.

[83] J. Kurlbaum and U. Frese, "A Benchmark Data Set for Data Association," University of Bremen, Tech. Rep. 017, 2009.

[84] J. L. Elman, "Finding Structure in Time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[85] K. Yu, K. Alexandr, and M. A. Sobolev, "Recurrent Neural Network and Extended Kalman Filter in SLAM Problem," in *3rd IFAC International Conference on Intelligent Control and Automation Science*, vol. 23, no. 3, September 2013, pp. 4–7.

[86] C. Van Pham and Y. N. Wang, "Robust Adaptive Trajectory Tracking Sliding Mode Control Based on Neural Networks for Cleaning and Detecting Robot Manipulators," *Journal of Intelligent & Robotic Systems*, vol. 79, no. 1, pp. 101–114, 2015.

[87] Z. Wei and S. X. Yang, "Neural Network Based Extended Kalman Filter for Localization of Mobile Robots," in *Proceedings of the 8th World Congress on Intelligent Control and Automation*, June 2011, pp. 937–942.

[88] A. Chatterjee and F. Matsuno, "A Neuro-Fuzzy Assisted Extended Kalman Filter-Based Approach for Simultaneous Localization and Mapping (SLAM) Problems," *IEEE Transactions on Fuzzy Systems*, vol. 15, no. 5, pp. 984–997, 2007.

[89] G. Dudek and M. Jenkin, *Computational Principles of Mobile Robotics*, 2nd ed. Cambridge University Press, 2010.

[90] Y. Bengio, P. Simard, and P. Frasconi, "Learning Long Term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[91] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-Propagating Errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[92] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simmulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[93] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," *Proceedings of the 27th International Conference on Machine Learning*, no. 3, pp. 807–814, 2010.

[94] L. E. Baum and T. Petrie, "Statistical Inference for Probabilistic Functions of Finite State Markov Chains," *The Annals of Mathematical Statistics*, vol. 37, no. 6, pp. 1554–1563, 1966.

[95] J. Blanco, "Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-range SLAM," University of Malaga, Spain, Tech. Rep., March 2008.

[96] M. Smith, I. Baldwin, W. Churchill, R. Paul, P. Newman, and L. D. Set, "The New College Vision and Laser Data Set," *The International Journal of Robotics Research*, vol. 28, no. 5, pp. 595–599, May 2009.

[97] D. Kamen, R. Ambrogi, R. Duggan, R. Heinzmann, B. Key, A. Skoskiewicz, and P. Kristal, "Human transporter," December 1997, US Patent 5,701,965.

[98] F. Michaud, D. Létourneau, J.-F. Paré, M.-A. Legault, R. Cadrin, M. Arsenault, Y. Bergeron, M.-C. Tremblay, F. Gagnon, M. Millette, P. Lepage, Y. Morin, and S. Caron, "Azimut, a Leg-track-wheel Robot," in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2003, pp. 2553–2558.

[99] F. Ferland, L. Clavien, J. Frémy, D. Létourneau, F. Michaud, and M. Lauria, "Teleoperation of AZIMUT-3, an Omnidirectional Non-holonomic Platform with Steerable Wheels," in *IEEE/RSJ*

2010 International Conference on Intelligent Robots and Systems, October 2010, pp. 2515–2516.

[100] J. Frémy, F. Ferland, L. Clavien, D. Létourneau, F. Michaud, and M. Lauria, "Force-controlled Motion of a Mobile Platform," *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, no. 1, pp. 2517–2518, October 2010.

[101] M. Labbe and F. Michaud, "Online Global Loop Closure Detection for Large-scale Multi-session Graph-based SLAM," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, September 2014, pp. 2661–2666.

[102] J. Gonzalez, J. L. Blanco, and F. A. Moreno-Duenas, "The Malaga Urban Dataset : High-rate Stereo and Lidars in a realistic urban scenario," *The International Journal of Robotics Research*, vol. 2013, pp. 1–11, 2013.

[103] M. N. H. Siddique and M. O. Tokhi, "Training Neural Networks: Backpropagation vs. Genetic Algorithms," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 4, July 2001, pp. 2673–2678.

[104] A. Howard and K. Nate, "Gazebo," http://gazebosim.org/, 2002.

[105] R. Smith, "Open dynamics engine," http://ode.org/, 2000.

[106] M. A. Sherman, A. Seth, and S. L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia IUTAM*, vol. 2, no. Supplement C, pp. 241 – 261, 2011, iUTAM Symposium on Human Body Dynamics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210983811000241

[107] P. Goodwin and R. Lawton, "On the asymmetry of the symmetric MAPE," *International Journal of Forecasting*, vol. 15, no. 3, 2, pp. 405–408, 1999.

[108] B. Horn, "Closed-form solution of absolute orientation using unit quaternions," *Journal of the Optical Society of America*, vol. 4, no. 4, pp. 629–642, April 1987.

[109] R. Kümmerle, B. Steder, C. Dornhege, M. Ruhnke, G. Grisetti, C. Stachniss, and A. Kleiner, "On Measuring the Accuracy of SLAM Algorithms," *Autonomous Robots*, vol. 27, no. 4, pp. 387–407, September 2009.

[110] L. T. DeCarlo, "On the Meaning and Use of Kurtosis," *Psychological Methods*, vol. 2, no. 3, pp. 292–307, 1997.

[111] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 4, November 1995, pp. 1942–1948.

[112] R. Eberhart and J. Kennedy, "A New Optimizer Using Particle Swarm Theory," in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, October 1995, pp. 39–43.

[113] J. R. Koza, "Genetic Programming as a Means for Programming Computers by Natural Selection," *Statistics and Computing*, vol. 4, no. 2, pp. 87–112, 1994.

[114] J. Lampinen and A. Vehtari, "Bayesian Approach for Neural Networks - Review and case studies." *Neural Network*, vol. 14, no. 3, pp. 257–274, 2001.

[115] L. K. Li, "Approximation Theory and Recurrent Networks," *International Joint Conference on Neural Networks*, vol. 2, no. 2, pp. 266–271, 1992.

[116] J. Tani and M. Ito, "Self-organization of Behavioral Primitives as Multiple Attractor Dynamics: A Robot Experiment," *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans*, vol. 33, no. 4, pp. 481–488, 2003.

[117] J. T. Lo and D. Bassu, "Mathematical Justification of Recurrent Neural Networks with Long- and Short-Term Memories," in *International Joint Conference on Neural Networks*, 1999, pp. 364–369.

[118] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to Forget: Continual Prediction with LSTM," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[119] D. Monner and J. A. Reggia, "A Generalized LSTM-like Training algorithm for Second-order Recurrent Neural Networks," *Neural Networks*, vol. 25, pp. 70–83, 2012.

[120] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[121] H. Geng, L. Wang, and P. Liu, "Dictionary Learning for Large-scale Remote Sensing Image Based on Particle Swarm Optimization," in *12th International Conference on Signal Processing (ICSP)*, 2014, pp. 784–789.

[122] J. D. Schaffer, R. A. Caruana, and L. J. Eshelman, "Using Genetic Search to Exploit the Emergent Behavior of Neural Networks," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1-3, pp. 244–248, 1990.

[123] L. Vanneschi, M. Castelli, and S. Silva, "A Survey of Semantic Methods in Genetic Programming," *Genetic Programming and Evolvable Machines*, vol. 15, no. 2, pp. 195–214, 2014.

[124] E. Galvan-Lopez, "Efficient Graph-based Genetic Programming Representation with Multiple Outputs," *International Journal of Automation and Computing*, vol. 5, no. 1, pp. 81–89, 2008.

[125] T. Stephens, "gplearn," https://github.com/trevorstephens/gplearn, 2016.

[126] D. Lowe, "Object Recognition from Local Scale-invariant Features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, September 1999, pp. 1150–1157.

[127] ——, "Distinctive Image Features from Scale-invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[128] H. Bay, T. Tuytelaars, and L. V. Gool, "SURF: Speeded Up Robust Features," in *Proceedings of the 9th European Conference on Computer Vision (ECCV)*, May 2006, pp. 404–417.

[129] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors With Automatic Algorithm Configuration," *Proceedings of the Fourth International Conference on Computer Vision Theory*

and Applications, pp. 331–340, 2009.

[130] C. Silpa-Anan and R. Hartley, "Optimised kd-Trees for Fast Image Descriptor Matching," *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 978–985, June 2008.

[131] A. Gil, Ó. Reinoso, Ó. M. Mozos, C. Stachniss, and W. Burgard, "Improving Data Association in Vision-based SLAM," in *IEEE International Conference on Intelligent Robots and Systems*, October 2006, pp. 2076–2081.

[132] G. Nebehay and R. Pflugfelder, "Consensus-based Matching and Tracking of Keypoints for Object Tracking," in *2014 IEEE Winter Conference on Applications of Computer Vision, WACV 2014*, March 2014, pp. 862–869.

[133] A. Gil, O. Reinoso, and M. Ballesta, "Managing Data Association in Visual SLAM using SIFT features," in *International Journal of Factory Automation Robotics and Soft Computing*, 2007, pp. 179–184.

[134] F. Lu and E. Milios, "Globally Consistent Range Scan Alignment for Environment Mapping," *Autonomous Robots*, vol. 4, no. 4, pp. 333–349, 1997.

ADDENDUM A LANDMARK DETECTION AND MATCHING

Many SLAM applications do not have prior knowledge of what features are available in an environment. The implication is that the vehicle needs to update the existing landmarks constantly in order to keep track of its location. As a result landmark matching and maintenance become a significant hurdle that needs to be addressed.

A.1 MATCHING LANDMARKS

Matching landmarks requires preprocessing of the image features in order to find their descriptors. These descriptors can be used to match the detected features in an image to the landmarks. The most common feature detector and descriptor algorithms used are the scale-invariant feature transform (SIFT) [126, 127] and speeded-up robust features (SURF) [128], each offering similar performance with regard to accuracy. However, SURF tends to be computationally less expensive and thus slightly faster.

The actual matching of one set of descriptors to another can be done using either a Brute-force or FLANN-based (Fast Library for Approximate Nearest Neighbours) matcher [129]. The difference is that the FLANN-based matcher creates kd-trees [130] that allow for fast comparison. Note that as with any kd-tree application, some accuracy is sacrificed for speed. Using these matchers, the current image's features can be matched to all the landmarks and their corresponding values can be measured. For implementation, both the matcher and detector algorithms available from OpenCV were used.

To increase the robustness of landmark matching, Lowe [126] proposed finding the nearest two matches for each feature descriptor and comparing the distance between the two landmarks. By calculating the ratio between the two descriptors and filtering any matches that do not meet the required ratio, many false positive matches can be eliminated from consideration. The ratio can consequently be varied between 0.7 and 0.25, depending on the environment.

A further matching criterion is to determine whether the location of the match is close to the actual landmark. A simple minimum Euclidean distance can therefore be stipulated that each match needs to adhere to. The minimum distance therefore ensures that minimal false positives are detected and limits the effect that any such false positives may have on the system as a whole. Algorithm A.1 provides the general procedure used to extract measurements from an image.

Algorithm A.1 MatchLandmarks(currImageData)

```
for all i in len(stableLandmarks) do
    matches = getFLANNMatches(stableDescriptor[i], currImageDescriptors)
    if matches.size() >= minGroupMatch then
        tgtPts, srcIdx, tgtIdx = extractImgKeyPts(matches, currImageKeypoints)
        for all m in matches do
            pt = depthToPoint(currDepthImage,tgtPts[p])
            globalPt = currPose ⊕ sensorTF ⊕ pt
            if isNear(globalPt, stableLandmarks[i]) then
                measurement[i] = calcRangebearing(pt)
                measIdx.append(i)
            end if
        end for
    end if
end for
    return measurements, measIdx
```

A.2 ADDING NEW LANDMARKS

Detecting new landmarks and adding them to the state is one of the largest problems faced during data association. Adding landmarks that are observed infrequently only increases the state while offering

no real benefit during localisation and mapping. Conversely, not including enough new landmarks can lead to the system losing track of the vehicle's pose.

The simplest method to add new landmarks is simply matching the previous image's features to the current one and then adding the newly detected landmarks. However, the number of landmarks added to the system quickly becomes untenable and computationally expensive. The occurrence ratio of the new landmarks is also not checked to determine their stability.

To increase robustness, a small subset of previous images can be used to determine if the descriptor occurs frequently, as described in [131]. Thus the landmarks added are at least assured of being detected frequently in the current observational window. To ensure that no duplicate landmarks are added to the state, the currently detected landmarks are also compared to all the landmarks in the state and any whose matching ratio is too close are removed.

The next challenge faced by landmark maintenance is deciding when to add the landmarks. Checking if new landmarks are detected at each new processed image is computationally expensive, especially since a number of previous images are used to ensure stability. A preferable approach would be to make use of the number of landmarks detected during observation. If only a few landmarks are detected, the likelihood of landmark sparsity is higher and future observations may not be able to detect any landmarks. Adding new landmarks just before this sparsity occurs ensures that new landmarks are available without unnecessarily increasing the number of landmarks in the state.

An additional improvement to landmarks handling concerns locations where they are detected. By noting that feature detectors commonly detect descriptors around objects (because of the edges, etc.) and that these points are relatively close to each other, the landmarks can be grouped using a consensus-based approach [132]. By specifying that any descriptors within a minimum Euclidean distance from each other belong to a certain landmark, a more descriptive representation of the environment can be generated with the landmark's location specified as the centroid of all the descriptors' points.

Thus each landmark will contain a number of descriptors to use during matching where a minimum ratio of descriptor matches needs to be present before the match is accepted. Moreover, a minimum number of descriptors needs to be present within each landmark in order to ensure stability. Another advantage of grouping landmarks is that other locations whose descriptors have a partial overlap

need not be removed from consideration. Algorithm A.2 provides the basic setup used to find stable descriptors in the current image and add the group to the stable groups of landmarks.

Algorithm A.2 UpdateLandmarks(currFeatureData)

```
1: tgtPts, tgtDesc = findStableDescriptors(featureDataMem)
2: for all p in tgtPts do
3:     pts = depthToPoint(depthImage,tgtPts[p])
4:     globalPt = currPose ⊕ sensorTF ⊕ pts
5:     tmpLandmarks.append( createTmpLandmark(globalPt, descriptor[p], tgtPts[p]) )
6: end for
7: for all m in range(0, tmpLandmarks.size()) do
8:     currGroup = tmpLandmark[m]
9:     for all n in range(m, tmpLandmarks.size()) do
10:        if isNear(tmpLandmark[m] ,tmpLandmark[n] ) then
11:            currGroup.append(tmpLandmark[n])
12:        end if
13:    end for
14:    if currGroup.size() >= minGroupSize then
15:        addStableLandmark(currGroup)
16:    end if
17: end for
```

A.3 REMOVING LANDMARKS

Computational overhead is a significant difficulty faced when using landmarks as the number of landmarks gradually increases while a vehicle traverses an environment. As a result many algorithms use a limiting factor to remove landmarks that become irrelevant to localisation and mapping.

One of the most common techniques employed is the moving window approach that removes landmarks from consideration based on the time they were added. The advantage of this approach is that a limit can be set on the number of landmarks that can be taken into consideration, thus limiting the computational overhead. One problem with the moving window approach is that if an environment is revisited, no loop-closure can occur, as the landmarks have been removed from memory.

An alternative approach is removing the landmarks based on their occurrence ratio [133]. By setting the minimum number of times that a landmark needs to be detected, any landmarks that are only matched infrequently are removed. The largest difficulty faced with this approach is that the most recent landmarks may not meet the minimum occurrence rate or that the rate is set too low. To eliminate the first problem one can limit the landmarks that are evaluated by specifying a percentage value (e.g. only the first 80% of the landmarks are evaluated).

Using the aforementioned approach, however, does not solve the problem of choosing an appropriate minimum occurrence rate. Thus many landmarks may be removed initially while during the later stages only a few are removed. To solve this problem, one can increase the minimum occurrence ratio as the algorithm progresses, thus allowing the algorithm to remove landmarks that become irrelevant. An alternative is to calculate the mean/median of the occurrences and to remove any landmarks whose occurrence rate is less than half of the average.

A last component that needs to be taken into account during landmark maintenance is when to remove the landmarks. If a fixed limit on the number of landmarks is set, the algorithm will eventually reach a stage where landmarks are constantly being removed. To overcome this problem, one can simply scale the number of landmarks allowed each time landmarks are removed. Algorithm A.3 provides the basic implementation of removing the landmarks as the vehicle progresses through an environment.

Algorithm A.3 removeLandmarksMean()

```
1: N = len(stableLandmarks)
2: mn = 1/N * sum_{i=0}^{N} numLandmarkObservations[i]
3: for i = 0 to N × 0.8 do
4:     if numLandmarkObservations[i] < mn × 0.5 then
5:         remove landmark's location, descriptor and occurrence rate
6:         remove corresponding rows and columns from the covariance matrix
7:     end if
8: end for
```

ADDENDUM B MOTION COMPOSITION

Data captured from a vehicle are relative to the sensor's current reference frame. Hence the data need to be converted from a local reference frame to the global frame using a homogeneous transformation, more commonly known as the motion composition operator (MCO) [134]. The MCO is commonly defined as:

$$\mathbf{p}_{sensor} = \mathbf{p}_{vehicle} \oplus \mathbf{p}_{relative}. \tag{B.1}$$

The easiest method for calculating data's global coordinates is first converting a vehicle's pose and the data into their corresponding 4×4 homogeneous matrices. The global pose of a vehicle is known to be:

$$\mathbf{p}_{vehicle} = [t_x, t_y, t_z, q_x, q_y, q_z, q_w], \tag{B.2}$$

where t_x, t_y, t_z is the translation from the origin to the vehicle and q_x, q_y, q_z, q_w are the quaternions describing the vehicle's heading. Assuming the quaternions have been normalised using:

$$q_{normalised} = \frac{[q_x, q_y, q_z, q_w]^T}{\sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}}. \tag{B.3}$$

the homogeneous matrix describing a vehicle transform from the origin can be calculated as:

$$\mathbf{H}_{vehicle} = \begin{bmatrix} q_w^2 + q_x^2 - qy^2 - q_z^2 & 2(qxq_y - q_w q_z) & 2(q_z q_y + q_r q_x) & t_x \\ 2(qxq_y + q_w q_z) & q_w^2 - q_x^2 + qy^2 - q_z^2 & 2(q_y q_z - q_r q_x) & t_y \\ 2(qzq_x - q_w q_y) & 2(qyq_z + q_w q_x) & q_w^2 - q_x^2 - qy^2 + q_z^2 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{B.4}$$

Similarly, the homogeneous coordinates for the sensor *relative* to the vehicle can be defined. If the sensor's location from the vehicle's origin can be defined as:

$$\mathbf{p}_{relative} = [t_{sx}, t_{sy}, t_{sz}, q_{sx}, q_{sy}, q_{sz}, q_{sw}], \tag{B.5}$$

its homogeneous transform can also be calculated. In order to determine the exact pose for the sensor, the two poses need to be composed. For homogeneous matrices this can be achieved through matrix multiplication, as shown in Equation B.6. Once calculated, the homogeneous matrix can be converted back into a 7D pose to describe the sensor's pose.

$$\mathbf{H}_{sensor} = \mathbf{H}_{vehicle}\mathbf{H}_{relative}. \tag{B.6}$$

Hence to describe the global position of any landmark data captured by the sensor, the data needs to be related to the global reference frame instead of the sensor's reference frame. The global position of the data thus can be defined as:

$$\mathbf{p}_{global} = \mathbf{p}_{local} \oplus (\mathbf{p}_{vehicle} \oplus \mathbf{p}_{relative}), \tag{B.7}$$

where $\mathbf{p}_{global}$ is the landmark's global position and $\mathbf{p}_{local}$ is the landmark's position detected by the sensor. Similarly, the landmarks' local position can be defined in terms of the landmark's global position (see Equation B.8). This is commonly referred to as the inverse motion composition operator (IMCO).

$$\mathbf{p}_{local} = \mathbf{p}_{global} \ominus (\mathbf{p}_{vehicle} \oplus \mathbf{p}_{relative}). \tag{B.8}$$

Calculating the IMCO can be achieved using homogeneous matrices. However, unlike the MCO, the matrices are multiplied using the homogeneous matrix's inverse, as shown below:

$$\mathbf{H}_{local} = \mathbf{H}_{global}(\mathbf{H}_{vehicle}\mathbf{H}_{relative})^{-1}. \tag{B.9}$$

ADDENDUM C DIFFERENTIAL DRIVE DYNAMICS

The following appendix explains how a differential drive vehicle's dynamics can be determined using Lagrange's Equations of motion. Figure C.1 details the general setup for a differential drive vehicle where:

- W is the distance from the center of the wheel to the center of the vehicle,
- $[X_W, Y_W, Z_W]$ are world coordinates and are referenced according to the origin O,
- N is the vehicle's inertial reference frame,
- $[X_R, Y_R, Z_R]$ are the vehicle's coordinates,
- θ is the vehicle's heading angle in relation to the world frame,
- t_w is the wheel thickness and r_w the wheel radius,
- ϕ_1 and ϕ_2 is the angular position of each wheel,
- m_b is the mass of the body,
- m_w is the mass of the wheels,
- I_{xx}, I_{yy}, I_{zz} is the moments of inertia of the body (assumed to be a cube).

Furthermore, the following assumptions are made regarding the differential drive:

1. The wheels have no slippage.
2. The body is symmetric (i.e. the wheels are symmetrically placed over the center of rotation).
3. Each wheel is in contact with the ground at a single point.

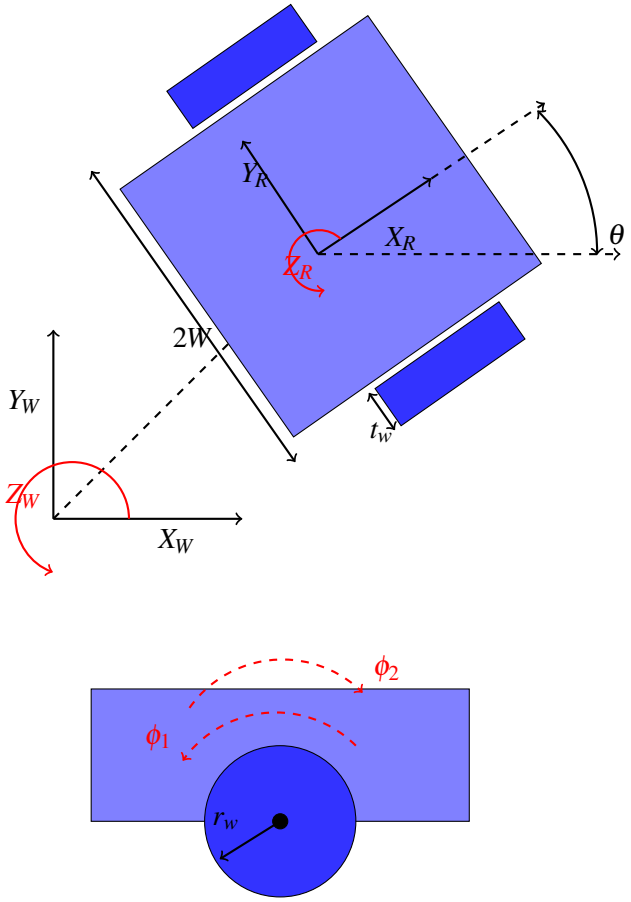


Figure C.1. Differential drive dynamic setup, using the world coordinates $[X_W, Y_W, Z_W]$ as reference.

C.1 KINEMATICS

Before the dynamics can be defined, the kinematics of the vehicle need to be taken into consideration. Specifically, the wheel velocity and non-holonomic constraints of the vehicle need to be described. Note that the kinematics used in the following section differ slightly from those provided in the main document. This is mainly due to the fact that the wheels' position is taken into account, which can be related to the vehicle's velocity.

C.1.1 Wheel velocities

To calculate the wheel velocities at a certain point (V_C), the velocity of each wheel hub (V_H) first needs to be calculated by making use of the linear (V_R) and angular velocities (ω_R) of the robot. In addition, the vector from the center of the body to the hub point (d_{H1}) and the relative orientation of the body to

the fixed reference frame (R) is required. The following equations provide each of the aforementioned requirements

$$\vec{V_R} = [\dot{x}, \dot{y}, 0]^T \tag{C.1}$$

$$\vec{\omega_R} = \left[0, 0, \dot{\theta}\right]^T \tag{C.2}$$

$$\vec{d_{H1}} = [0, -W, 0]^T \tag{C.3}$$

$$R = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{C.4}$$

The velocity of the wheel hub can therefore be defined as:

$$\begin{aligned} \vec{V_{H1}} &= \vec{V_R} + \vec{\omega_R} \times R\vec{d_{H1}} \\ &= \begin{bmatrix} \dot{x} + W\dot{\theta}\cos(\theta) \\ \dot{y} + W\dot{\theta}\sin(\theta) \\ 0 \end{bmatrix}. \end{aligned} \tag{C.5}$$

Similarly, the velocity of the second wheel hub can be calculated to produce:

$$\vec{V_{H2}} = \begin{bmatrix} \dot{x} - W\dot{\theta}\cos(\theta) \\ \dot{y} - W\dot{\theta}\sin(\theta) \\ 0 \end{bmatrix}. \tag{C.6}$$

Once the velocity of the wheel hub is known, the wheel's contact point velocity can be calculated by making use of the angular velocity of the wheel (ω_{W1}) and the vector from the hub's center to the contact point (d_{H1C1}).

$$\vec{\omega_{W1}} = \dot{\phi}_1 \begin{bmatrix} \sin(\theta) \\ -\cos(\theta) \\ 0 \end{bmatrix} \tag{C.7}$$

$$\vec{d_{H1C1}} = [0, 0, r_w]^T. \tag{C.8}$$

The wheel velocities can then be defined as:

$$\begin{aligned}\vec{V_C} =& \vec{V_H} + \vec{\omega_W} \times \vec{d_{HC}} \\ \Rightarrow \vec{V_{C1}} =& \begin{bmatrix} \dot{x} + W\dot{\theta}\cos(\theta) + r_w\dot{\phi_1}\cos\theta \\ \dot{y} + W\dot{\theta}\sin(\theta) + r_w\dot{\phi_1}\cos\theta \\ 0 \end{bmatrix} \qquad \text{(C.9)} \\ \Rightarrow \vec{V_{C2}} =& \begin{bmatrix} \dot{x} - W\dot{\theta}\cos(\theta) - r_w\dot{\phi_2}\cos\theta \\ \dot{y} - W\dot{\theta}\sin(\theta) - r_w\dot{\phi_2}\cos\theta \\ 0 \end{bmatrix}. \qquad \text{(C.10)}\end{aligned}$$

C.1.2 Non-holonomic constraints

The assumption that no wheel-slip occurs during motion means that the velocities at the contact points of the wheels are zero ($\vec{V_{C1}} = 0, \vec{V_{C2}} = 0$). Using this assumption and assuming that the wheel displacement ($\dot{\phi_1}$ and $\dot{\phi_2}$) is known, four simultaneous equations can be derived from Equation C.9 and Equation C.10 with three unknowns ($\dot{x}, \dot{y}, \dot{\theta}$). Rearranging the equations leads to:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \frac{r_w}{2} \begin{bmatrix} (\dot{\phi_2} - \dot{\phi_1})\cos(\theta) \\ (\dot{\phi_2} - \dot{\phi_1})\sin(\theta) \\ -\dfrac{(\dot{\phi_2} + \dot{\phi_1})}{W} \end{bmatrix}, \qquad \text{(C.11)}$$

which can also be expressed as :

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \frac{r_w}{2} \begin{bmatrix} (\dot{\phi_2} - \dot{\phi_1}) \\ 0 \\ -\dfrac{(\dot{\phi_2} + \dot{\phi_1})}{W} \end{bmatrix}. \qquad \text{(C.12)}$$

C.2 DYNAMICS

The Lagrange's equations of motion can be defined as:

$$L(q_i, \dot{q}_i) = KE(q_i, \dot{q}_i) - PE(q_i), \tag{C.13}$$

where $(\vec{q})$ is the Lagrange state vector that contains the generalised coordinates. For a differential drive the state vector is defined as:

$$\vec{q} = [x, y, \theta, \phi_1, \phi_2]^T. \tag{C.14}$$

Commonly, these system are assumed to be unconstrained. Hence the equations of motion as the vehicle moves can be calculated by differentiating the Lagrangian, as shown in Equation C.15.

$$\frac{d}{dt}\frac{\partial L(\vec{q}, \dot{\vec{q}})}{\partial \dot{q}_i} - \frac{\partial L(\vec{q}, \dot{\vec{q}})}{\partial q_i} = \vec{Q}, \tag{C.15}$$

where $\vec{Q}$ are the generalised forces acting on the system. As a differential drive system has a number of constraints due to the vehicle's structure, the constraints must be included in the calculations by using Lagrange multipliers. These non-holonomic constraints are described using Equation C.12 as:

$$C(q)\dot{\vec{q}} = 0 \tag{C.16}$$

$$C(q) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & \frac{r_w}{2} & \frac{-r_w}{2} \\ -\sin(\theta) & \cos(\theta) & 0 & 0 & 0 \\ 0 & 0 & 1 & \frac{r_w}{2W} & \frac{-r_w}{2W} \end{bmatrix}. \tag{C.17}$$

Using these results, Lagrange's equations take the form:

$$\frac{d}{dt}\left[\frac{\partial L(q_i, \dot{q}_i)}{\partial q}\right] - \frac{\partial L(q_i, \dot{q}_i)}{\partial q} - C(q)^T \lambda = T, \tag{C.18}$$

where λ is the vector of undetermined Langrange multipliers.

C.2.1 Kinetic Energy

The total kinetic energy can then be defined using each individual body's linear velocity and angular velocity at their centroids (center of mass) using:

$$KE_{total} = \frac{1}{2}\sum_{i=1}^{n}\left(m_i\|V_i\|^2 + \boldsymbol{\omega_i}^T \mathbf{I_i}\boldsymbol{\omega_i}\right). \tag{C.19}$$

As the velocity of the main body is related to the distance to its center of mass, an assumption can be made that the velocity of the body is along the fixed reference frame's x-axis of distance (d_b). The body's velocity can then be defined as:

$$\vec{V_B} = \vec{V_R} + \omega_R \times \vec{r_{cm_b}} \tag{C.20}$$

$$= \begin{bmatrix} \dot{x} \\ \dot{y} \\ 0 \end{bmatrix} + \dot{\theta} \times \begin{bmatrix} d_b\cos(\theta) \\ d_b\sin(\theta) \\ 0 \end{bmatrix}. \tag{C.21}$$

Thus the kinetic energy of the main body is:

$$KE_{body} = \frac{m_b}{2}\left(\dot{x}^2 + \dot{y}^2 + d_b^2\dot{\theta}^2 + 2d_b\dot{\theta}\left(\dot{y}\cos(\theta) - \dot{x}\sin(\theta)\right)\right) + \frac{I_b}{2}\dot{\theta}^2. \tag{C.22}$$

Calculating the kinetic energy of the wheels follows a similar approach, where the hub is assumed to be the center of mass. This means that the hub velocities (Equation C.5 and Equation C.6) can be used to calculate the kinetic energy. Note that the wheel's angular velocity rotates about an axis that is normal to the plane. Thus the angular velocities for both wheels are:

$$\Omega_{w1} = \begin{bmatrix} \dot{\phi_1}\sin(\theta) \\ \dot{\phi_1}\cos(\theta) \\ \dot{\theta} \end{bmatrix} \tag{C.23}$$

$$\Omega_{w2} = \begin{bmatrix} \dot{\phi_2}\sin(\theta) \\ \dot{\phi_2}\cos(\theta) \\ \dot{\theta} \end{bmatrix}. \tag{C.24}$$

Furthermore, if the wheel is assumed to be a solid cylinder, the wheel's inertia matrix can easily be calculated. Hence each wheel's inertia is:

$$I_w = \begin{bmatrix} I_{wxx} & 0 & 0 \\ 0 & I_{wyy} & 0 \\ 0 & 0 & I_{wzz} \end{bmatrix} = \begin{bmatrix} \frac{m_w}{12}(3r_w^2 + t_w^2) & 0 & 0 \\ 0 & \frac{m_w r_w^2}{2} & 0 \\ 0 & 0 & \frac{m_w}{12}(3r_w^2 + t_w^2) \end{bmatrix}, \tag{C.25}$$

and the subsequent kinetic energy for the wheels can be defined as:

$$KE_{w1} = \frac{m_w}{2}\left(\dot{x}^2 + \dot{y}^2 + W^2\dot{\theta}^2 + 2W\dot{\theta}\left(\dot{x}\cos(\theta) - \dot{y}\sin(\theta)\right)\right) + \frac{I_{wzz}}{2}\dot{\theta}^2 + \frac{I_{wyy}}{2}{\dot{\phi}_1}^2 \tag{C.26}$$

$$KE_{w2} = \frac{m_w}{2}\left(\dot{x}^2 + \dot{y}^2 - W^2\dot{\theta}^2 + 2W\dot{\theta}\left(\dot{x}\cos(\theta) - \dot{y}\sin(\theta)\right)\right) + \frac{I_{wzz}}{2}\dot{\theta}^2 + \frac{I_{wyy}}{2}{\dot{\phi}_1}^2. \tag{C.27}$$

The total kinetic energy for a differential drive is therefore:

$$KE_{total} = \frac{m_T}{2}\left(\dot{x}^2 + \dot{y}^2\right) + m_b d_b \dot{\theta}\left(\dot{y}\cos(\theta) - \dot{x}\sin(\theta)\right) + \frac{I_T}{2}\dot{\theta}^2 + \frac{I_{wyy}}{2}\left({\dot{\phi}_1}^2 + {\dot{\phi}_2}^2\right). \tag{C.28}$$

where

$$m_T = m_b + 2m_w \tag{C.29}$$

$$I_T = I_b + m_b d_b^2 + 2m_w W^2 + 2I_{wzz}. \tag{C.30}$$

C.3 LAGRANGE'S EQUATIONS OF MOTION

Substituting Equation C.28 into Equation C.13 and taking their derivatives (from Equation C.18) results in the following terms (note the difference between $\frac{\partial L}{\partial \dot{q}}$ and $\frac{\partial L}{\partial q}$) :

$$\frac{\partial L}{\partial \dot{q}} = \begin{bmatrix} m_T\dot{x} - m_b d_b \dot{\theta}\sin(\theta) \\ m_T\dot{y} + m_b d_b \dot{\theta}\cos(\theta) \\ I_T\dot{\theta} + m_b d_b(\dot{y}cos(\theta) - \dot{x}\sin(\theta)) \\ I_{wyy}p\dot{h}i_1 \\ I_{wyy}\dot{\phi}_2 \end{bmatrix} \tag{C.31}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} = M(q) = m_b d_b \dot{\theta} \begin{bmatrix} m_T & 0 & -m_b d_b \sin(\theta) & 0 & 0 \\ 0 & m_T & m_b d_b \cos(\theta) & 0 & 0 \\ -m_b d_b \sin(\theta) & -m_b d_b \cos(\theta) & I_T & 0 & 0 \\ 0 & 0 & 0 & I_{wyy} & 0 \\ 0 & 0 & 0 & 0 & I_{wyy} \end{bmatrix} \tag{C.32}$$

$$\frac{\partial L}{\partial q} = B(q,\dot{q}) = m_b d_b \dot{\theta} \begin{bmatrix} 0 \\ 0 \\ \dot{y}sin(\theta) - \dot{x}\cos(\theta) \\ 0 \\ 0 \end{bmatrix}. \tag{C.33}$$

Using the above terms results in an equation of the form:

$$M(q)\ddot{q} + B(q,\dot{q}) - C(q)^T\vec{\lambda} = T. \tag{C.34}$$

where:

$$C(\dot{q})^T\vec{\lambda} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \\ \frac{r_w}{2} & 0 & \frac{r_w}{2W} \\ \frac{-r_w}{2} & 0 & \frac{-r_w}{2W} \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix}, \tag{C.35}$$

and:

$$T = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \tau_1 \\ \tau_2 \end{bmatrix}. \tag{C.36}$$

Thus the variables τ_1 and τ_2 are the torques applied to each of the wheels. To determine the Lagrange multipliers ($\vec{\lambda}$) of the dynamics, the constraint $C(q)(\dot{q}) = 0$ must be satisfied (see non-holonomic constraints). These constraints must therefore also satisfy the following:

$$\frac{d}{dt}\left[C(q)(\dot{q})\right] = 0 \tag{C.37}$$

$$\Rightarrow C(q)(\ddot{q}) + \dot{C}(q)\dot{q} = 0 \tag{C.38}$$

which can be used to solve for $\vec{\lambda}$ by using Equation C.34 and $\ddot{q}$. However, it should be clear that solving for $\vec{\lambda}$ is a lengthy process involving inverting matrices (and ensuring that the matrices are invertible). For completeness the actual result is provided below.

$$\vec{\lambda} = -\left[C(q)M(q)^{-1}C(q)^T\right]^{-1}\left[C(q)M(q)^{-1}\Big(T - B(q,\dot{q})\Big) + \dot{C}(q)(\dot{q})\right]. \tag{C.39}$$

As a result of the complexity the following **assumptions** are made in order to ensure that formulas remain tractable:

1. Let the main body's center of mass lie along the wheel axes, i.e. $d_b = 0$. Using this simplification means that $B(q,\dot{q}) = 0$ and that $M(q)$ becomes a diagonal matrix.
2. Evaluate the dynamical equations at $\theta = 0$. While not a simplification, this choice allows some of the terms to be eliminated, thus simplifying the process of solving $\vec{\lambda}$.

These simplifications reduce the calculations to:

$$\vec{\lambda} = -\left[C(q)M(q)^{-1}C(q)^T\right]^{-1}\left[C(q)M(q)^{-1}T + \dot{C}(q)\dot{q}\right], \tag{C.40}$$

where:

$$C(q)M(q)^{-1}C(q)^T = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & m_T^{-1} & 0 \\ 0 & 0 & \beta \end{bmatrix} \tag{C.41}$$

$$C(q)M(q)^{-1}T = \frac{r_w}{2} \begin{bmatrix} \tau_1 - \tau_2 \\ 0 \\ \dfrac{\tau_1 + \tau_2}{W} \end{bmatrix} I_{wyy}^{-1} \tag{C.42}$$

$$\dot{C}(q)\dot{q} = \dot{\theta} \begin{bmatrix} \dot{x}\sin(\theta) - \dot{y}\cos(\theta) \\ \dot{x}\cos(\theta) + \dot{y}\sin(\theta) \\ 0 \end{bmatrix}, \tag{C.43}$$

and:

$$\alpha = m_T^{-1} + 2I_{wyy}^{-1}\left(\frac{r_w}{2}\right)^2 \tag{C.44}$$

$$\beta = I_T^{-1} + 2I_{wyy}^{-1}\left(\frac{r_w}{2}\right)^2, \tag{C.45}$$

Substituting these results into Equation C.40 and evaluating at $\theta = 0$ produces:

$$\vec{\lambda} = \frac{r_w}{2} I_{wyy}^{-1} \begin{bmatrix} \alpha^{-1}(\tau_1 - \tau_2) \\ 0 \\ \beta^{-1}\dfrac{\tau_1 + \tau_2}{W} \end{bmatrix} + \dot{\theta} \begin{bmatrix} -\alpha(\dot{y}) \\ m_T\dot{x} \\ 0 \end{bmatrix}, \tag{C.46}$$

which can then be used in Equation C.34 to provide the system's dynamics. Thus the full equation for each of the terms $Mq\ddot{q}, B(q,\dot{q})$ and $C(q)^T\vec{\lambda}$ is provided below. This equation can then be used to describe the dynamical motion of a differential drive vehicle.

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ \tau_1 \\ \tau_2 \end{bmatrix} = m_b d_b \dot{\theta} \begin{bmatrix} m_T \dot{x} - m_b d_b \ddot{\theta} \sin(\theta) \\ m_T \ddot{y} + m_b d_b \ddot{\theta} \cos(\theta) \\ -m_b d_b \sin(\theta) \ddot{x} - -m_b d_b \cos(\theta) \ddot{y} + I_T \ddot{\theta} \\ I_{wyy} \ddot{\phi_1} \\ I_{wyy} \ddot{\phi_2} \end{bmatrix} + m_b d_b \dot{\theta} \begin{bmatrix} 0 \\ 0 \\ \dot{y} sin(\theta) - \dot{x} \cos(\theta) \\ 0 \\ 0 \end{bmatrix}$$

$$- \begin{bmatrix} \cos(\theta) \left(\frac{r_w}{2} I_{wyy}^{-1} \alpha^{-1} (\tau_1 - \tau_2) - \dot{\theta} \alpha(\dot{y}) \right) \\ -\sin(\theta) \left(\frac{r_w}{2} I_{wyy}^{-1} \alpha^{-1} (\tau_1 - \tau_2) - \dot{\theta} \alpha(\dot{y}) \right) \\ \frac{r_w}{2} I_{wyy}^{-1} \beta^{-1} \frac{\tau_1 + \tau_2}{W} + \dot{\theta} m_T \dot{x} \\ \frac{r_w^2}{4} I_{wyy}^{-1} \alpha^{-1} (\tau_1 - \tau_2) - \frac{r_w}{2} \dot{\theta} \alpha(\dot{y}) + \frac{r_w^2}{4W} I_{wyy}^{-1} \beta^{-1} \frac{\tau_1 + \tau_2}{W} + \frac{r_w}{2W} \dot{\theta} m_T \dot{x} \\ -\frac{r_w^2}{4} I_{wyy}^{-1} \alpha^{-1} (\tau_1 - \tau_2) + \frac{r_w}{2} \dot{\theta} \alpha(\dot{y}) - \frac{r_w^2}{4W} I_{wyy}^{-1} \beta^{-1} \frac{\tau_1 + \tau_2}{W} - \frac{r_w}{2W} \dot{\theta} m_T \dot{x} \end{bmatrix}. \quad \text{(C.47)}$$

ADDENDUM D ROS AND GAZEBO

The following appendix provides implementation details concerning the evaluation framework used during the course of the work. Note that the system was implemented in ROS Indigo and Gazebo 2.2.

D.1 ROS

The following section provides additional information on the ROS implementation. In particular, the evaluation framework is described along with the SLAM algorithms and datasets that can be used with the framework.

D.1.1 Framework software

An evaluation framework was created in ROS to easily add SLAM algorithms, datasets as well as Gazebo simulations. To achieve this, the framework had to automatically launch the correct ROS nodes and link their respective topics for communication. In addition, the navigation stack and vehicle model controllers needed to be generated and connected on request. Once execution was finished the framework evaluated the trajectory and maps created and provided a summary of the results. Furthermore, the algorithms needed to be compared to one another in order to gauge their performance.

To allow the framework to perform this as autonomously as possible, the framework not only needed to open the correct ROS nodes, but also close them once a particular test was finished. Hence the framework executed an algorithms' corresponding launch files, with additional command line

parameters for the specific dataset's topics and implementation. The framework was built using Python and depended mainly on the standard ROS packages such as the sensor, geometry and navigation messages. The framework also depended on the point cloud libraries and OpenCV to convert any images and point-clouds for processing. Lastly the framework depended on the Gazebo libraries to link the simulated vehicle models to ROS.

D.1.2 Configuration files

To facilitate ease of use as well as flexibility, most of the implemented software made use of the YAML Ain't Markup language (YAML) format. In particular, the main configuration file was was set up to allow the framework to define multiple tests. The following provides an example of the main configuration file that was used to start execution:

```
tests:
  navigationTestRTABMAP:
    Skip: [ 1 , 2, 5, 10 ]
    VehicleModels: pioneer  #Used in Gazebo
    Dataset: Azimut      #or Freiburg, New College, Malaga
    Algorithm: RTABMAP  #or DIFFEKFSLAM, modelLearn, Gmapping, etc.
    Evaluation:
      ATE: true
      RPE: false
      genPoseGraph: true
      mapType: occupancyGrid
      storeResults: true
      showResults: false
    Environment: freiburgRobot  #Used in Gazebo
    Navigation:   #Used in Gazebo
      Type: Simple
      Skip: 10
```

In most of the cases, each of the parameters defined in the main configuration file had their own configuration files that provided additional setup parameters. Hence depending on the parameters selected by a test, different configurations could automatically be loaded. In the example, the evaluation framework was set up to evaluate one of the Azimut datasets using RTABMAP. In addition, the test also conducted simulations in Gazebo with a vehicle model for comparison (see Section D.2.2 for more information).

D.1.3 Datasets

A number of datasets could be loaded automatically by the evaluation framework. Furthermore, the framework could automatically connect a particular dataset's topics to an algorithm, as long as the correct data was available. Currently, the following datasets can be loaded into system:

- The Freiburg Robot datasets,
- The New College Datasets,
- The Malaga Urban datasets,
- The Azimut-3 multi-session datasets.

As the Azimut and Freiburg datasets are available as ROS *Bag-files* loading them was implemented by sending command line arguments to a new terminal. The New College and Malaga datasets' data, in comparison, were loaded during execution. Hence in each case an implementation was created that published the raw data onto ROS topics, with their corresponding sensor transforms. For these datasets a clock server also needed to be created that handled ROS's internal time and synchronised the data.

D.1.4 SLAM algorithms

A number of SLAM algorithms were setup in the evaluation framework. The following lists all of the SLAM algorithms that were connected in the evaluation framework:

- Odometry EKF-SLAM,
- Kinematic model differential drive EKF-SLAM,

- Learned motion model's EKF-SLAM,
- HectorSLAM,
- GMapping,
- RTABMAP [101].

In general, the execution procedure for each algorithm is described by Algorithm D.1. The dataset handler, as the name suggests, loads all of the parameters required by the dataset and returns a dictionary containing the topics that the actual data will be published on. From this, the algorithm can be set up to subscribe to the correct topics while providing information on the topics where the results are published. These topics are subscribed by the evaluation framework and used to evaluate the results once the algorithm has finished execution. Lastly, the paths that the files were stored to were returned and used when a multiple tests were conducted. As such a comparison of different tests and algorithms could automatically be generated.

Algorithm D.1 Execution procedure for the algorithms

```
1: dsetTopics = datasetHandler.loadDataset(dsetName)
2: topicDictionary = execHandler.loadAlgorithm(algName,dsetTopics, dsetName )
3: topicSubscriber = dataSubscriber.poseSubscriber(topicDictionary)
4: topicSubscriber.subscribeTopics()
5: datasetHandler.execDataset()
6: while execHandler.isNodeRunning(algName) do
7:     Wait until dataset is done
8: end while
9: poses = poseHandler.loadGTPoses(dsetName)
10: resultHandler.processAlgorithmResults(topicSubscriber, poses)
11: storedPosePath, storedMapPath = resultHandler.getStoredFilesPath()
        return storedPosePath, storedMapPath
```

D.2 GAZEBO

The following section provides a brief overview the Gazebo implementation along with the creation of the vehicle models.

D.2.1 Vehicle Control

Controlling a vehicle in Gazebo is commonly implemented using the vehicle's joint state or velocity controllers. As such the PID control for any joint needs to be defined and a controller implemented to that can control the vehicle. However, a Gazebo-ROS plugin for a differential drive is available in ROS to provide control. Thus the plugin was used to control differential drive and skid-steer vehicles instead of creating a new controller.

D.2.1.1 Gazebo-ROS differential drive controller

The differential drive's controller used to control a vehicle is based on the kinematic model of a differential drive. The controller could therefore also be used to control a skid-steer drive vehicle by specifying that both front and rear wheels received the same control commands. In addition, a number of limitations could be specified to more accurately represent the vehicle's dynamics. The following list some of the parameters that can be used to model the vehicle's motion and set additional limitations:

- The wheel separation,
- Wheel diameter,
- Velocity limits,
- Acceleration limits,
- The update rate.

All of the aforementioned limitations was defined in a configuration file that was loaded in a launch file. The launch file also loaded the vehicle's unified robot description format (URDF) file into Gazebo, launched the velocity controller, robot's state publisher and a subscriber node. The state publisher shared information of the vehicle's state between ROS and Gazebo, while the subscriber node listened for incoming velocity commands that the velocity controller used to drive the vehicle. Specifically, the velocity controller made use of linear and angular velocities to control the vehicle.

D.2.1.2 Vehicle Model Generator

Creating a simple vehicle model that works in ROS and Gazebo was done using the URDF format. The URDF file describes the vehicle's physical properties such as geometry, weight, inertia, friction coefficients and links to other components. Additionally, any movable joints and sensors needs to be defined along with their corresponding hardware interfaces and implementation. As such, the more complex a vehicle becomes, the more one needs to keep track of the exact names used for each component to ensure consistency across all the files. Furthermore, if multiple different vehicles are used, one needs to ensure consistent naming conventions across all the vehicles if one wants to incorporate any vehicle in an implementation that automatically loads these vehicles.

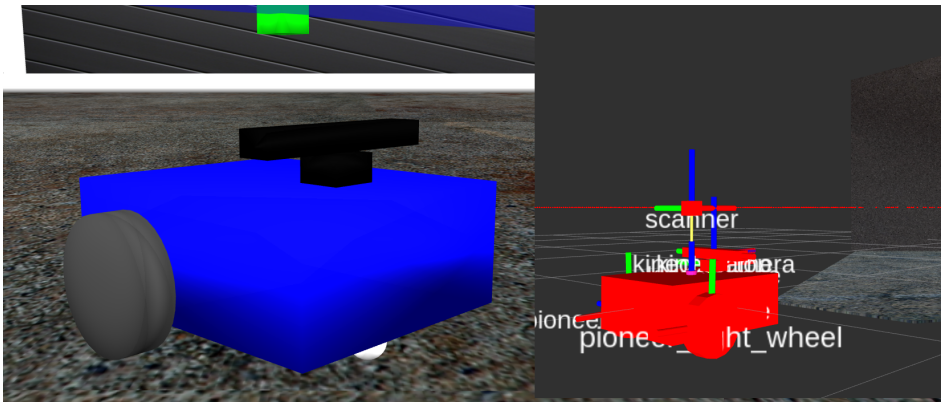
Thus, it was decided to create a vehicle model generator that could automatically generate the configuration, launch and URDF files while maintaining consistency. To accomplish this, the YAML format was used to define the basic structure of a vehicle, the sensors and type of controller used. From this file, all the vehicle's required files could be generated consistently. An example of the basic setup used to generate an entire vehicle is shown below. Note that the YAML file was set up to generate multiple vehicles using a single file.

```
vehicles:
  pioneer:
    NumWheels: 2
    InitialPose: [ 0, 0, 0.5, 0.0, 0.0, 0 ]
    Controller:
      Type: Differential
      UpdateRate: 20
      hasVelocityLimits: False
      pCovariance: [0.05, 0.05, 1.0, 100.0, 100.0, 0.078]
      tCovariance: [0.05, 0.05, 1.0, 100.0, 100.0, 0.078]
    Base:
      Dimensions: [0.455, 0.381, 0.175]
      Mass: 7
      Colour: Blue
    Wheels:
```

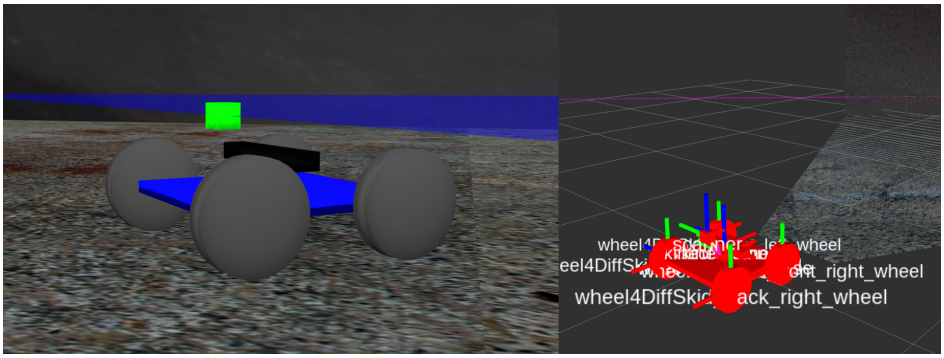
```
    Dimensions: [0.0975, 0.05]
    Mass: 1.5
    Friction: [0.5, 0.5]
    Transform: [-0.1, 0.2155, -0.042, 1.57, 0, 3.14]
    Colour: DarkGrey
  Sensors:
    Kinect:
      Parent: Base
      Transform: [ 0.1, 0, 0.0, 0, 0, 0]
      Colour: Black
    Laser:
      Colour: Green
      Parent: Base
      Transform: [ 0.0, 0, 0.2, 0, 0, 0]
      UpdateRate: 40
      Scan:
        Samples: 720
        Resolution: 1
        Angle: 1.5707
      Range:
        Min: 0.1
        Max: 20.0
        Resolution: 0.1
    IMU:
      Parent: Base
      Transform: [ 0.0, 0.1, 0.0, 0, 0, 0]
```

While the system could only generate relatively simple vehicle models, it does ensure smooth operation during testing as the naming conventions are standardised for all vehicles. Currently, 3 types of sensors can be included in the generated vehicle models: A Kinect, a laser scanner and an IMU. Furthermore, two types of controllers currently available for automatic generation is a differential and skid-steer driven system. Figure D.1 provides an illustration of a the generated vehicles in both Gazebo and

RViz.



(a) Pioneer robot with front caster.



(b) Skid-steer drive vehicle.

Figure D.1. Gazebo (left) and Rviz (right) visualisation of the generated vehicle models.

D.2.2 Navigation

ROS's navigation stack was used to control the vehicle's movement. Specifically, the *move_base* package was used to calculate the control commands sent to the vehicles. The navigation stack allowed the system to have repeatable control of the vehicle for various tests. In addition, the control could be applied automatically, thus requiring no input from the user.

This allowed the system to compare different vehicles using the same exact control commands. Furthermore, the vehicles could be compared to real-world datasets as long as a simulated environment could be generated that matched the real-world environment. Two such environments were created that roughly approximated the Freiburg dataset's environment and a part of the Azimut multi-session

datasets' environment. Algorithm D.2 provides the general execution procedure used to compare the performance of the simulated and real-world vehicles under such circumstances.

Algorithm D.2 Gazebo-Algorithm execution tests

```
for each defined test do
    Load parameters
    Execute algorithm using dataset
    Save results to file
    Unload algorithm node
    Load Gazebo environment
    for all vehicleModels in test do
        Load vehicle model and Navigation stack
        Execute algorithm using Gazebo topics
        Compare vehicle tests to dataset's results
        Store all test results
        Unload vehicle model, navigation nodes and algorithm nodes
    end for
    Unload Gazebo
end for
```

ADDENDUM E ADDITIONAL EXPERIMENTAL RESULTS

E.1 MODEL LEARNING TESTS (TANH ACTIVATION)

The model learning tests were also conducted for TDL-NNs with a $\tanh(\cdot)$ activation. Neural nets with and without control were also tested as well as if an increase in initial poses provided any marked improvements. Table E.1 the general parameters used during these tests.

Table E.1. Parameters for the linear activation function tests.

Parameter	**Value**	**Parameter**	**Value**
NN structure	{5, 7, 11, 15, 29, 45 }	Activation functions	tanh
Discontinuity split	Yes	Test set	Simulated test sets
Training sets (simulated)	[Arc, linear, rotational, stationary]	NN approach	[Memory, memory with control]
Training sets (Freiburg)	[Robot SLAM1, Robot SLAM2, Robot SLAM3]	Test set (Freiburg)	Robot 360
Number of previous states	{ 3, }	Increasing training data	[Initial poses]
Scaling factor (simulated) (x, y, θ)	[30.0, 30.0, 30.0]	Scaling factor (simulated) (v_f, ω)	[30.0, 30.0]

E.1.1 Experimental results

E.1.1.1 Freiburg datasets

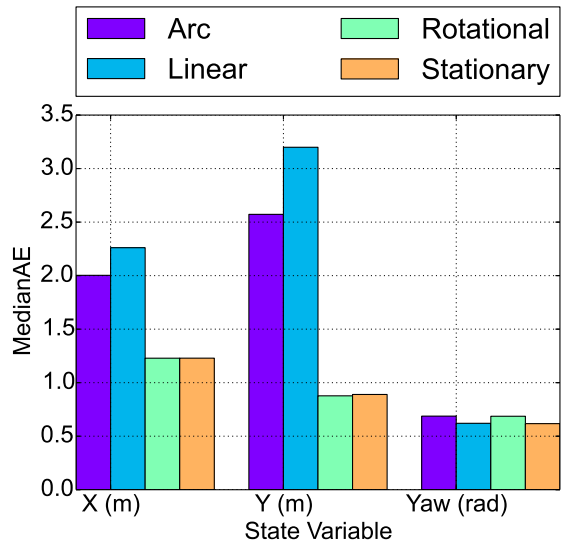
The following section provides the results for the TDL-NN implementation using a $\tanh(\cdot)$ activation function using the model learning metrics discussed in Chapter 4. Figure E.1 provides the overall errors obtained for a $\tanh(\cdot)$ activation function using a memory of 3 and 15 hidden nodes. The skew and kurtosis values observed were significantly less than those observed with the linear activation functions. In addition, arc and linear motion's positional values the skew and kurtosis values were closer to a Gaussian distribution. However, the AE_{median} and $AE_{medianAD}$ errors were significantly higher compared to the NNs with linear activation function that were trained with the Freiburg datasets.

Analysis of the effects of the network's structure on the NNs performance revealed that different network sizes did have an impact on the AE_{median} error (see Table E.2). Even though the NNs performed better with one state variable, it was usually counteracted by worse performance in another. A typical example of such a case is with a NN with 29 hidden nodes. While the x-estimate was significantly less for linear and arc motion, the yaw-estimates were higher than what was observed in other networks. However, the best value found were still significantly higher than the NNs with linear activation functions.

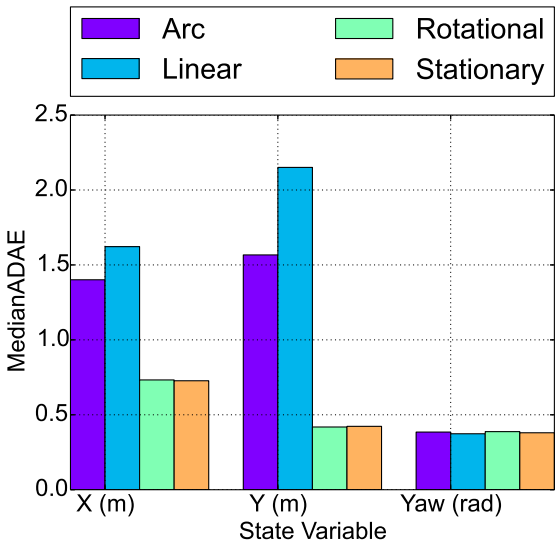
E.1.1.2 Simulated Datasets

The previous results indicated that the TDL-NNs trained with the Freiburg datasets performed significantly worse than the corresponding NNs trained with linear activation functions. However, because variations on the errors were observed the TDL-NNs were trained with the simulated datasets to determine if any improvements could be observed. In addition, NNs with control inputs included were also trained to establish if the control had similar effect to those observed for the linear activation functions.

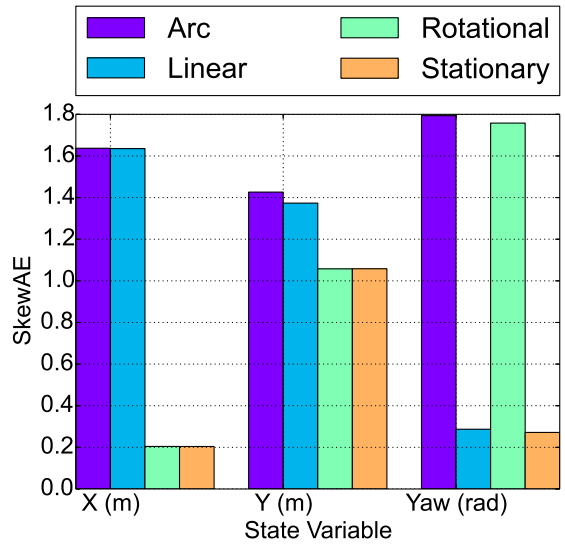
The results observed, however, indicated that the simulated datasets performed slightly worse than the Freiburg datasets (see Table E.3). In addition, no very little variations were observed over different network structures. Similarly, additional control inputs had little observable effect on the network's



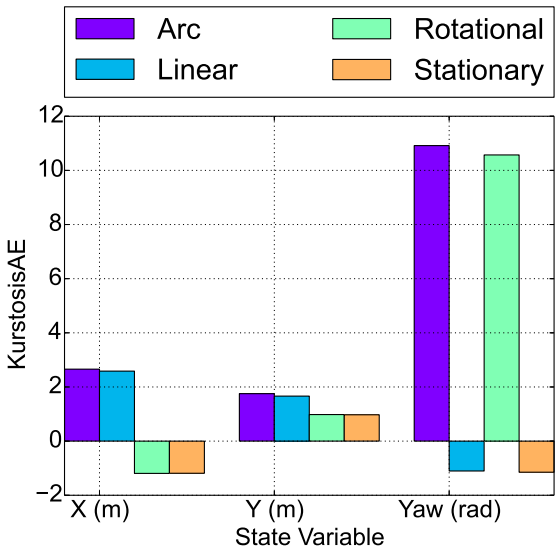
(a) AE_{median}.



(b) $AE_{medianAD}$.



(c) AE_{skew}.



(d) $AE_{kurtosis}$.

Figure E.1. Bar graphs of each motion type's error for a NN with 15 hidden nodes and a memory of 3. The x-, y- and yaw errors were each graphed next to each other for direct comparison.

performance. One possible cause for the degradation in performance of the TDL-NNs is the scaling values, which, for the simulated datasets was required to be 30. As a result, the control variables may not have a significant impact during training. Nonetheless, TDL-NNs with a $\tanh(\cdot)$ activation function could not produce similar result to the linear activation function.

Table E.2. The overall AE_{median} error (measured in meters and radians) for a TDL-NN's trained with the Freiburg datasets.

	NN structure			
Test Data	**7**	**15**	**29**	**45**
Arc motion (x,y,θ)	2.052 2.623 0.915	1.996, 2.660, 0.502	1.446, 2.500, 1.412	2.299, 2.741, 1.138
Linear motion (x,y,θ)	2.326, 3.270, 0.802	2.281, 3.297, 0.540	1.424, 3.085, 1.370	2.552, 3.393, 0.859
Rotational motion (x,y,θ)	1.282, 0.847, 0.936	1.210, 0.928, 0.473	1.480, 1.004, 1.547	1.394, 1.099, 1.196
Stationary motion (x,y,θ)	1.287, 0.859, 0.864	1.214, 0.951, 0.477	1.517, 1.020, 1.421	1.408, 1.110, 0.975

Lastly, the effects of increasing the amount of initial poses were tested. However, doubling the number of initial poses used by the training data resulted in no observable improvements. As with the linear activation tests, the AE_{median} errors remained almost identical to the previous result.

Table E.3. The overall AE_{median} error for a TDL-NN's trained with the arc motion simulated datasets using 15 initial poses.

	NN structure (No Control)			**NN structure (Control)**		
Test Data	**5**	**11**	**45**	**5**	**11**	**45**
	15 initial poses					
Arc motion (x,y,θ)	2.174 2.787 1.210	2.179 2.749 1.192	2.222 2.816 1.216	2.254 2.723 1.189	2.221 2.812 1.212	2.224 2.813 1.215
Linear motion (x,y,θ)	2.438 3.433 1.046	2.438 3.400 0.970	2.489 3.479 0.992	2.494 3.384 0.921	2.484 3.474 0.988	2.489 3.477 0.992
Rotational motion (x,y,θ)	1.281 1.068 1.220	1.357 1.111 1.198	1.384 1.131 1.216	1.353 1.030 1.204	1.382 1.130 1.212	1.384 1.135 1.216
Stationary motion (x,y,θ)	1.286 1.078 0.991	1.364 1.113 0.972	1.389 1.134 0.992	1.353 1.925 0.980	1.386 1.131 0.988	1.389 1.136 0.992
	28 initial poses					
Arc motion (x,y,θ)	2.187 2.792 1.214	2.216 2.804 1.213	2.223 2.813 1.221	2.196 2.811 1.194	2.225 2.814 1.210	2.217 2.809 1.223
Linear motion (x,y,θ)	2.451 3.454 0.987	2.476 3.463 0.988	2.488 3.478 0.994	2.458 3.471 0.982	2.486 3.473 0.987	2.483 3.475 0.997
Rotational motion (x,y,θ)	1.375 1.127 1.211	1.376 1.132 1.212	1.383 1.135 1.218	1.369 1.137 1.201	1.382 1.138 1.210	1.382 1.135 1.221
Stationary motion (x,y,θ)	1.387 1.129 0.988	1.380 1.134 0.989	1.388 1.138 0.994	1.379 1.136 0.984	1.386 1.139 0.988	1.387 1.137 0.998

ADDENDUM F SIMULATED DATASETS

The subsequent sections provide the poses and controls used to generate the training and test sets used in the model learning tests.

F.1 TEST SETS

Table F.1. Test poses used during the model learning methodology

Pose #	X (m)	Y (m)	θ (rad)	Pose #	X (m)	Y (m)	θ (rad)
1	1.8	-0.7	-0.78	7	0.8	-1.2	-1.78
2	3.2	2.8	0.78	8	2.6	-1.5	2.34
3	0.6	1.3	1.578	9	-0.6	-1.1	-0.3
4	0.0	0.0	0.0	10	0.0	0.5	1.1
5	1.2	1.2	0.5	11	2.4	1.3	2.4
6	2.1	0.0	-0.9	12	-1.6	0.2	-1.9

Table F.2. Test controls used to by the model learning methodology

	Arc Motion		**Linear Motion**	
Control #	V_f **(m/s)**	ω **(rad/s)**	V_f **(m/s)**	ω **(rad/s)**
1	0.2	0.2	0.1	0.0
2	0.2	-0.2	0.25	0.0
3	0.4	0.12	0.4	0.0
4	0.4	0.12	0.6	0.0
5	0.7	0.08	0.8	0.0
6	0.7	0.08	0.9	0.0
7	1.1	0.05	1.2	0.0
8	1.1	0.05	1.5	0.0
9	1.5	0.15	1.7	0.0
10	1.5	0.15	1.9	0.0
11	2.0	0.2	2.2	0.0
12	2.0	-0.2	2.4	0.0
	Rotational Motion		**Stationary Motion**	
	V_f **(m/s)**	ω **(rad/s)**	V_f **(m/s)**	ω **(rad/s)**
1	0.0	0.03	0.0	0.0
2	0.0	0.07	0.001	0.0
3	0.0	0.12	0.0	0.001
4	0.0	0.15	0.0	-0.001
5	0.0	0.19	0.001	-0.001
6	0.0	0.23	0.001	-0.001
7	0.0	-0.03		
8	0.0	-0.07		
9	0.0	-0.12		
10	0.0	-0.15		
11	0.0	-0.19		
12	0.0	-0.23		

F.2 TRAINING DATASETS

Table F.3. The first 56 poses used to generate the simulated datasets

Pose #	X (m)	Y (m)	θ (rad)	Pose #	X (m)	Y (m)	θ (rad)
1	1.8	-0.7	-0.78	29	0.0	0.0	-1.2
2	-1.8	0.7	-0.78	30	0.85	0.4	-0.5
3	1.8	-0.7	0.78	31	-0.85	0.4	0.5
4	-1.8	0.7	0.78	32	0.85	-0.4	0.5
5	3.2	2.8	0.78	33	-0.85	-0.4	-0.5
6	-3.2	-2.8	0.78	34	2.6	2.1	1.8
7	3.2	2.8	-0.78	35	-2.6	2.1	-1.8
8	-3.2	-2.8	-0.78	36	2.6	-2.1	-1.8
9	0.6	-1.3	1.578	37	-2.6	-2.1	-1.8
10	-0.6	1.3	1.578	38	1.6	1.3	0.9
11	0.6	-1.3	-1.578	39	-1.6	1.3	0.9
12	-0.6	1.3	-1.578	40	1.6	-1.3	-0.9
13	1.2	0.3	2.78	41	-1.6	-1.3	-0.9
14	-1.2	-0.3	-2.78	42	3.1	1.6	2.4
15	1.2	0.3	-2.78	43	-3.1	1.6	2.4
16	-1.2	-0.3	2.78	44	3.1	-1.6	-2.4
17	-2.3	1.2	1.24	45	-3.1	-1.6	-2.4
18	2.3	1.2	1.24	46	0.5	3.3	0.0
19	-2.3	1.2	-1.24	47	-0.5	3.3	0.0
20	2.3	1.2	-1.24	48	0.5	-3.3	0.0
21	-0.3	2.6	-0.24	49	-0.5	-3.3	0.0
22	0.3	2.6	0.24	50	2.4	0.2	1.3
23	-0.3	2.6	0.24	51	-2.4	0.2	1.3
24	0.3	2.6	-0.24	52	2.4	-0.2	-1.3
25	0.0	0.0	0.0	53	2.4	-0.2	-1.3
26	0.0	0.0	0.48	54	1.3	2.4	2.9
27	0.0	0.0	-0.48	55	-1.3	2.4	2.9
28	0.0	0.0	1.2	56	1.3	-2.4	-2.9

Table F.4. The last 57 poses used to generate the simulated datasets

Pose #	X (m)	Y (m)	θ (rad)	Pose #	X (m)	Y (m)	θ (rad)
57	-1.3	-2.4	-2.9	86	0.85	0.4	-2.1
58	1.8	-0.7	-1.9	87	-0.85	0.4	2.1
59	-1.8	0.7	-1.9	88	0.85	-0.4	-2.1
60	1.8	-0.7	1.9	89	-0.85	-0.4	-2.1
61	1.8	-0.7	1.9	90	2.6	2.1	0.6
62	3.2	2.8	2.4	91	-2.6	2.1	0.6
63	-3.2	-2.8	2.4	92	2.6	-2.1	-0.6
64	3.2	2.8	-2.4	93	-2.6	-2.1	-0.6
65	-3.2	-2.8	-2.4	94	1.6	1.3	1.3
66	0.6	-1.3	0.578	95	-1.6	1.3	1.3
67	-0.6	1.3	0.578	96	1.6	-1.3	-1.3
68	0.6	-1.3	-0.587	97	-1.6	-1.3	-1.3
69	-0.6	1.3	-0.587	98	3.1	1.6	0.9
70	1.2	0.3	0.0	99	-3.1	-1.6	0.9
71	-1.2	-0.3	0.0	100	3.1	-1.6	-0.9
72	-1.2	0.3	0.0	101	-3.1	-1.6	-0.9
73	1.2	-0.3	0.0	102	0.5	3.3	1.7
74	-2.3	1.2	2.8	103	-0.5	3.3	1.7
75	2.3	1.2	2.8	104	0.5	-3.3	1.7
76	-2.3	1.2	-2.8	105	-0.5	-3.3	1.7
77	2.3	1.2	-2.8	106	2.4	0.2	2.3
78	-0.3	2.6	-1.4	107	-2.4	0.2	2.3
79	0.3	2.6	1.4	108	2.4	-0.2	-2.3
80	-0.3	2.6	1.4	109	2.4	-0.2	-2.3
81	0.3	2.6	-1.4	110	1.3	2.4	2.2
82	0.0	0.0	1.7	111	-1.3	2.4	2.2
83	0.0	0.0	-1.7	112	1.3	-2.4	-2.2
84	0.0	0.0	2.3	113	-1.3	-2.4	-2.2
85	0.0	0.0	-2.3				

Table F.5. Arc motion control for the simulated datasets (1-56)

Control #	V_f (m/s)	ω (rad/s)	Control #	V_f (m/s)	ω (rad/s)
1	0.05	0.03	29	0.2	0.11
2	0.05	-0.03	30	0.2	-0.11
3	0.05	0.06	31	0.2	0.14
4	0.05	-0.06	32	0.2	-0.14
5	0.05	0.09	33	0.2	0.17
6	0.05	-0.09	34	0.2	-0.17
7	0.05	0.11	35	0.2	0.19
8	0.05	-0.11	36	0.2	-0.19
9	0.05	0.14	37	0.2	0.22
10	0.05	-0.14	38	0.2	-0.22
11	0.05	0.17	39	0.2	0.25
12	0.05	-0.17	40	0.2	-0.25
13	0.05	0.19	41	0.2	0.27
14	0.05	-0.19	42	0.2	-0.27
15	0.05	0.22	43	0.2	0.29
16	0.05	-0.22	44	0.2	-0.29
17	0.05	0.25	45	0.35	0.03
18	0.05	-0.25	46	0.35	-0.03
19	0.05	0.27	47	0.35	0.06
20	0.05	-0.27	48	0.35	-0.06
21	0.05	0.29	49	0.35	0.09
22	0.05	-0.29	50	0.35	-0.09
23	0.2	0.03	51	0.35	0.11
24	0.2	-0.03	52	0.35	-0.11
25	0.2	0.06	53	0.35	0.14
26	0.2	-0.06	54	0.35	-0.14
27	0.2	0.09	55	0.35	0.17
28	0.2	-0.09	56	0.35	-0.17

Table F.6. Arc motion control for the simulated datasets (57-112)

Control #	V_f (m/s)	ω (rad/s)	Control #	V_f (m/s)	ω (rad/s)
57	0.35	0.17	85	0.5	0.25
58	0.35	-0.17	86	0.5	-0.25
59	0.35	0.19	87	0.5	0.27
60	0.35	-0.19	88	0.5	-0.27
61	0.35	0.22	89	0.5	0.29
62	0.35	-0.22	90	0.5	-0.29
63	0.35	0.25	91	0.7	0.03
64	0.35	-0.25	92	0.7	-0.03
65	0.35	0.27	93	0.7	0.06
66	0.35	-0.27	94	0.7	-0.06
67	0.35	0.29	95	0.7	0.09
68	0.35	-0.29	96	0.7	-0.09
69	0.5	0.03	97	0.7	0.11
70	0.5	-0.03	98	0.7	-0.11
71	0.5	0.06	99	0.7	0.14
72	0.5	-0.06	100	0.7	-0.14
73	0.5	0.09	101	0.7	0.17
74	0.5	-0.09	102	0.7	-0.17
75	0.5	0.11	103	0.7	0.19
76	0.5	-0.11	104	0.7	-0.19
77	0.5	0.14	105	0.7	0.22
78	0.5	-0.14	106	0.7	-0.22
79	0.5	0.17	107	0.7	0.25
80	0.5	-0.17	108	0.7	-0.25
81	0.5	0.19	109	0.7	0.27
82	0.5	-0.19	110	0.7	-0.27
83	0.5	0.22	111	0.7	0.29
84	0.5	-0.22	112	0.7	-0.29

Table F.7. Arc motion control for the simulated datasets (113-168)

Control #	V_f **(m/s)**	ω **(rad/s)**	**Control #**	V_f **(m/s)**	ω **(rad/s)**
113	0.9	0.03	141	1.0	0.11
114	0.9	-0.03	142	1.0	-0.11
115	0.9	0.06	143	1.0	0.14
116	0.9	-0.06	144	1.0	-0.14
117	0.9	0.09	145	1.0	0.17
118	0.9	-0.09	146	1.0	-0.17
119	0.9	0.11	147	1.0	0.19
120	0.9	-0.11	148	1.0	-0.19
121	0.9	0.14	149	1.0	0.22
122	0.9	-0.14	150	1.0	-0.22
123	0.9	0.17	151	1.0	0.25
124	0.9	-0.17	152	1.0	-0.25
125	0.9	0.19	153	1.0	0.27
126	0.9	-0.19	154	1.0	-0.27
127	0.9	0.22	155	1.0	0.29
128	0.9	-0.22	156	1.0	-0.29
129	0.9	0.25	157	1.15	0.03
130	0.9	-0.25	158	1.15	-0.03
131	0.9	0.27	159	1.15	0.06
132	0.9	-0.27	160	1.15	-0.06
133	0.9	0.29	161	1.15	0.09
134	0.9	-0.29	162	1.15	-0.09
135	1.0	0.03	163	1.15	0.11
136	1.0	-0.03	164	1.15	-0.11
137	1.0	0.06	165	1.15	0.14
138	1.0	-0.06	166	1.15	-0.14
139	1.0	0.09	167	1.15	0.17
140	1.0	-0.09	168	1.15	-0.17

Table F.8. Arc motion control for the simulated datasets (169-224)

Control #	V_f **(m/s)**	ω **(rad/s)**	**Control #**	V_f **(m/s)**	ω **(rad/s)**
169	1.15	0.17	197	1.3	0.25
170	1.15	-0.17	198	1.3	-0.25
171	1.15	0.19	199	1.3	0.27
172	1.15	-0.19	200	1.3	-0.27
173	1.15	0.22	201	1.3	0.29
174	1.15	-0.22	202	1.3	-0.29
175	1.15	0.25	203	1.45	0.03
176	1.15	-0.25	204	1.45	-0.03
177	1.15	0.27	205	1.45	0.06
178	1.15	-0.27	206	1.45	-0.06
179	1.15	0.29	207	1.45	0.09
180	1.15	-0.29	208	1.45	-0.09
181	1.3	0.03	209	1.45	0.11
182	1.3	-0.03	210	1.45	-0.11
183	1.3	0.06	211	1.45	0.14
184	1.3	-0.06	212	1.45	-0.14
185	1.3	0.09	213	1.45	0.17
186	1.3	-0.09	214	1.45	0-.17
187	1.3	0.11	215	1.45	0.19
188	1.3	-0.11	216	1.45	-0.19
189	1.3	0.14	217	1.45	0.22
190	1.3	-0.14	218	1.45	-0.22
191	1.3	0.17	219	1.45	0.25
192	1.3	-0.17	220	1.45	-0.25
193	1.3	0.19	221	1.45	0.27
194	1.3	-0.19	222	1.45	-0.27
195	1.3	0.22	223	1.45	0.29
196	1.3	-0.22	224	1.45	-0.29

Table F.9. Arc motion control for the simulated datasets (225-280)

Control #	V_f **(m/s)**	ω **(rad/s)**	**Control #**	V_f **(m/s)**	ω **(rad/s)**
225	1.6	0.03	253	1.75	0.11
226	1.6	-0.03	254	1.75	-0.11
227	1.6	0.06	255	1.75	0.14
228	1.6	-0.06	256	1.75	-0.14
229	1.6	0.09	257	1.75	0.17
230	1.6	-0.09	258	1.75	-0.17
231	1.6	0.11	259	1.75	0.19
232	1.6	-0.11	260	1.75	-0.19
233	1.6	0.14	261	1.75	0.22
234	1.6	-0.14	262	1.75	-0.22
235	1.6	0.17	263	1.75	0.25
236	1.6	-0.17	264	1.75	-0.25
237	1.6	0.19	265	1.75	0.27
238	1.6	-0.19	266	1.75	-0.27
239	1.6	0.22	267	1.75	0.29
240	1.6	-0.22	268	1.75	-0.29
241	1.6	0.25	269	1.9	0.03
242	1.6	-0.25	270	1.9	-0.03
243	1.6	0.27	271	1.9	0.06
244	1.6	-0.27	272	1.9	-0.06
245	1.6	0.29	273	1.9	0.09
246	1.6	-0.29	274	1.9	-0.09
247	1.75	0.03	275	1.9	0.11
248	1.75	-0.03	276	1.9	-0.11
249	1.75	0.06	277	1.9	0.14
250	1.75	-0.06	278	1.9	-0.14
251	1.75	0.09	279	1.9	0.17
252	1.75	-0.09	280	1.9	-0.17

Table F.10. Arc motion control for the simulated datasets (281-337)

Control #	V_f **(m/s)**	ω **(rad/s)**	**Control #**	V_f **(m/s)**	ω **(rad/s)**
281	1.9	0.17	309	2.05	0.25
282	1.9	-0.17	310	2.05	-0.25
283	1.9	0.19	311	2.05	0.27
284	1.9	-0.19	312	2.05	-0.27
285	1.9	0.22	313	2.05	0.29
286	1.9	-0.22	314	2.05	-0.29
287	1.9	0.25	315	2.2	0.03
288	1.9	-0.25	316	2.2	-0.03
289	1.9	0.27	317	2.2	0.06
290	1.9	-0.27	318	2.2	-0.06
291	1.9	0.29	319	2.2	0.09
292	1.9	-0.29	320	2.2	-0.09
293	2.05	0.03	321	2.2	0.11
294	2.05	-0.03	322	2.2	-0.11
295	2.05	0.06	323	2.2	0.14
296	2.05	-0.06	324	2.2	-0.14
297	2.05	0.09	325	2.2	0.17
298	2.05	-0.09	326	2.2	-0.17
299	2.05	0.11	328	2.2	0.19
300	2.05	-0.11	329	2.2	-0.19
301	2.05	0.14	330	2.2	0.22
302	2.05	-0.14	331	2.2	-0.22
303	2.05	0.17	332	2.2	0.25
304	2.05	-0.17	333	2.2	-0.25
305	2.05	0.19	334	2.2	0.27
306	2.05	-0.19	335	2.2	-0.27
307	2.05	0.22	336	2.2	0.29
308	2.05	-0.22	337	2.2	-0.29

Table F.11. Arc motion control for the simulated datasets (338-393)

Control #	V_f **(m/s)**	ω **(rad/s)**	**Control #**	V_f **(m/s)**	ω **(rad/s)**
338	2.35	0.03	366	2.5	0.11
339	2.35	-0.03	367	2.5	-0.11
340	2.35	0.06	368	2.5	0.14
341	2.35	-0.06	369	2.5	-0.14
342	2.35	0.09	370	2.5	0.17
343	2.35	-0.09	371	2.5	-0.17
344	2.35	0.11	372	2.5	0.19
345	2.35	-0.11	373	2.5	-0.19
346	2.35	0.14	374	2.5	0.22
347	2.35	-0.14	375	2.5	-0.22
348	2.35	0.17	376	2.5	0.25
349	2.35	-0.17	377	2.5	-0.25
350	2.35	0.19	378	2.5	0.27
351	2.35	-0.19	379	2.5	-0.27
352	2.35	0.22	380	2.5	0.29
353	2.35	-0.22	381	2.5	-0.29
354	2.35	0.25	382	2.65	0.03
355	2.35	-0.25	383	2.65	-0.03
356	2.35	0.27	384	2.65	0.06
357	2.35	-0.27	385	2.65	-0.06
358	2.35	0.29	386	2.65	0.09
359	2.35	-0.29	387	2.65	-0.09
360	2.5	0.03	388	2.65	0.11
361	2.5	-0.03	389	2.65	-0.11
362	2.5	0.06	390	2.65	0.14
363	2.5	-0.06	391	2.65	-0.14
364	2.5	0.09	392	2.65	0.17
365	2.5	-0.09	393	2.65	-0.17

Table F.12. Arc motion control for the simulated datasets (394-427)

Control #	V_f **(m/s)**	ω **(rad/s)**	**Control #**	V_f **(m/s)**	ω **(rad/s)**
394	2.65	0.17	411	2.8	-0.09
395	2.65	-0.17	412	2.8	0.11
396	2.65	0.19	413	2.8	-0.11
397	2.65	-0.19	414	2.8	0.14
398	2.65	0.22	415	2.8	-0.14
399	2.65	-0.22	416	2.8	0.17
400	2.65	0.25	417	2.8	-0.17
401	2.65	-0.25	418	2.8	0.19
402	2.65	0.27	419	2.8	-0.19
403	2.65	-0.27	420	2.8	0.22
404	2.65	0.29	421	2.8	-0.22
405	2.65	-0.29	422	2.8	0.25
406	2.8	0.03	423	2.8	-0.25
407	2.8	-0.03	424	2.8	0.27
408	2.8	0.06	425	2.8	-0.27
409	2.8	-0.06	426	2.8	0.29
410	2.8	0.09	427	2.8	-0.29

Table F.13. The linear, rotational and stationary motion control used to generate the simulated datasets

	Linear Motion		**Rotational Motion**		**Stationary Motion**	
Control #	V_f **(m/s)**	ω **(rad/s)**	V_f **(m/s)**	ω **(rad/s)**	V_f **(m/s)**	ω **(rad/s)**
1	0.25	0.0	0.0	0.05	0.0	0.0
2	0.6	0.0	0.0	0.11	0.001	0.0
3	0.85	0.0	0.0	0.16	0.0	0.001
4	1.15	0.0	0.0	0.27	0.0	-0.001
5	1.35	0.0	0.0	-0.05	0.001	-0.001
6	1.8	0.0	0.0	-0.11	0.001	-0.001
7	2.1	0.0	0.0	-0.16		
8	2.6	0.0	0.0	-0.27		