NTNU
Norwegian University of
Science and Technology

Deep Learning Metocean Simulation and its Applications in Marine Simulation-based Design

Ole Johan Jørgensen Lønnum

Marine Technology
Submission date: May 2018
Supervisor: Bjørn Egil Asbjørnslett, IMT
Co-supervisor: Endre Sandvik, IMT

Norwegian University of Science and Technology
Department of Marine Technology

MASTER'S THESIS IN MARINE TECHNOLOGY

SPRING 2018

for Stud. techn.

OLE JOHAN JØRGENSEN LØNNUM

"Deep Learning Metocean Simulation and its Applications in Marine Simulation-based Design"

Background

There is a distinct trend within marine systems design to move away from traditional static design methods to more dynamic methods to understand value-driving aspects during early design stages (Fathi et al., 2013). Traditional design methods are unable to cope with increasingly complex operational profiles since these methods normally depend on simplified calculations based on an average or a few representative conditions (Hagen and Grimstad, 2010; Fathi et al., 2013).

State-of-the-art design methods aims to understand the needs of the stakeholders and generate value-robust designs (Gaspar, 2013). Simulation-based design is one such state-of-the-art method which focuses on optimization over a wide range of conditions, thus creating a system that can deliver value over changing contexts (Fathi et al., 2013).

Simulation-based design approaches generally consists of an operation model combined with metocean data to produce a realistic description of the operating conditions in the temporal and spatial domain (Fathi et al., 2013). The availability and quality of metocean data therefore becomes an integral part of simulation-based design.

A tremendous amount of data on wind and sea state conditions are collected each day, but it remains relatively sparse to the size of the ocean space (Monbet et al., 2007). Hence, metocean simulation models are necessary to produce multiple realistic scenarios for simulation-based design applications.

Metocean simulation is most appropriately viewed in the context of stochastic metocean generators, which are statistical models whose realizations statistically match observed patterns (Hering et al., 2015).

Deep learning methods have dramatically improved the state-of-the-art in speech recognition, natural language processing, image recognition and many other domains (Krizhevsky et al., 2012; LeCun et al., 2015). Several studies have shown that deep learning models improve prediction performance for metocean data compared to traditional methods, but the potential of deep learning models as stochastic metocean generators in simulation-based design remains currently unknown (Monbet et al., 2007).

Objective
The objective of the master's thesis is to validate the quality of deep learning models as stochastic metocean generators and their practicality in a simulation-based design context.

Scope of Work
The candidate should presumably cover the following main points:

1. Describe relevant aspects of metocean data with regards to simulation-based design.
2. Describe and compare current methodologies for metocean simulation.
3. Present an introduction to deep learning with a focus on methods for sequence processing.
4. Develop stochastic generators for metocean simulation:
    a. Traditional stochastic generators based on current methodologies.
    b. Stochastic generator based on state-of-the-art deep learning methods.
5. Develop a realistic case study for validation of the stochastic generators developed in (4) with a primary focus on:
    a. Quality of results.
    b. Practicality in a simulation-based design context.
6. Discuss and conclude on the quality of deep learning stochastic metocean generators and their practicality in simulation-based design, based on the case study developed in (5).

Modus Operandi
Professor Bjørn Egil Asbjørnslett and PhD candidate Endre Sandvik will be the supervisors at NTNU. The work shall follow the NTNU guidelines for master's thesis work. The workload shall correspond to 30 ECTS credits.

Bjørn Egil Asbjørnslett
Professor/Main Supervisor

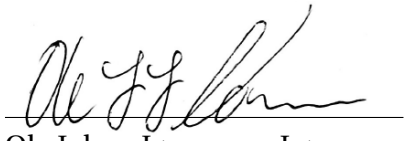
Preface

This master's thesis is part of the Master of Science degree in Marine Technology with a specialization in Marine Systems Design at the Norwegian University of Science and Technology, Trondheim. The thesis was written in its entirety during the spring of 2018, and the work load is equivalent to 30 ECTS. The focus of the thesis is on validation of the quality of deep learning models as stochastic metocean generators and their practicality in a simulation-based design context.

The master thesis builds on the preceding project thesis written in the fall of 2017 with the topic "Stochastic Wind and Sea State Simulators for Simulation-based Design Applications".

I would like to thank Professor Bjørn Egil Asbjørnslett for guiding me towards the topic of metocean simulation and PhD candidate Endre Sandvik for helpful guidance, interesting ideas and feedback throughout the process of writing this thesis.

Trondheim, June 11, 2018



Ole Johan Jørgensen Lønnum

"Machine Learning" by xkcd.com (reprint approved for non-merchandizing publications)

Abstract

The objective of the master's thesis was to validate the quality of deep learning models as stochastic metocean generators and practicality in the context of simulation-based design. The availability and quality of metocean data is an integral part of simulation-based design and stochastic metocean generators are necessary in many applications to produce synthetic metocean series. Deep learning has shown great results in many domains, but the potential of deep learning stochastic metocean generators is currently unknown. This was the foundation for the first research question: *How does the quality of a deep learning stochastic metocean generator compare to simpler traditional stochastic metocean generators?* The trade-off between practicality and quality of results in simulation-based design was investigated through the second research question: *Can deep learning models be practically used as stochastic metocean generators in simulation-based design applications?*

To answer the research questions, a new type of deep learning model called stochastic autoregressive long short-term memory (SAR-LSTM) network model was developed. The SAR-LSTM model was validated in a case study against traditional stochastic metocean generators: Markov chains, VAR and VARMA models. The case study consisted of simulating and matching selected statistical criteria of two bivariate series of significant wave height H_s and peak period T_p. The metocean series were first made approximately Gaussian by a lognormal Rosenblatt transformation introduced in this thesis. Then, a seasonal transformation which removed the yearly periodicity induced by the annual meteorological cycle was applied to make the series stationary. The resulting residuals were then simulated using the different model types before being back-transformed and validated.

The results showed that the SAR-LSTM network model did not adequately match the statistical criteria. The VAR and VARMA models performed exceptionally well in matching the statistical criteria and were found to give a sufficient description of H_s and T_p. The Markov chain models performed better than the SAR-LSTM model, but were subpar to the VAR and VARMA models. Thus, the conclusion to the first research question was that the SAR-LSTM network models produce results of a much lower quality than traditional stochastic metocean generators. The results further showed that the lognormal transformation introduced in this thesis should be used in favor of the commonly applied Box-Cox transformation if matching the joint distribution of H_s and T_p is of importance in the application.

The SAR-LSTM models were found to be unpractical due to the significant time and computational cost for implementing a single stochastic generator. Thus, the conclusion to the second research question was that deep learning stochastic metocean generators are not practical in simulation-based design, unless only a very limited set of unique metocean generators are required. The other models types were found to be practical in simulation-based design.

The VAR and VARMA models were found to be the best choice of models since they had the highest quality of results and were found to be highly practical. The extension from VAR to VARMA model produced better results on some criterion, but came at the price of a higher computational cost for estimating the model parameters.

Sammendrag

Målsetningen med denne masteroppgaven var å validere kvaliteten til dyp læring modeller som stokastiske havgeneratorer og deres praktikalitet i en simuleringsbasert designkontekst. Tilgjengligheten og kvaliteten på oseanografisk data er en essensiell del av simuleringsbasert design og stokastiske havgeneratorer er nødvendig i mange applikasjoner for å produsere syntetiske tidsserier. Dyp læring har vist gode resultater i mange domener, men potensialet til dyp læring stokastiske havgeneratorer er enda ukjent. Dette var grunnlaget for det første forskningsspørsmålet: *Hvordan er kvaliteten til en dyp læring stokastisk havgenerator sammenlignet med enklere tradisjonelle stokastiske havgeneratorer.* Avveiningen mellom praktikalitet og kvaliteten på resultater ble undersøkt gjennom det andre forskningsspørsmålet: *Kan dyp læring modeller praktisk brukes som stokastiske havgeneratorer i simuleringsbasert design applikasjoner.*

For å svare på forskningsspørsmålene ble en ny type dyp læring modell utviklet, kalt stokastisk autoregressiv lang korttidshukommelse (SAR-LSTM). SAR-LSTM modellen ble validert i en casestudie mot tradisjonelle stokastiske havgeneratorer: Markovkjeder, VAR og VARMA modeller. Casestudien bestod av å simulere og gjenskape utvalgte statiske kriterier for to bivariate serier bestående av signifikant bølgehøyde H_s og bølgetopp periode T_p. Havseriene ble først tilnærmet Gaussiske ved en lognormal Rosenblatt transformasjon introdusert i denne oppgaven. En sesong transformasjon som fjernet den årlige periodisiteten indusert av den årlige metrologiske syklusen ble deretter brukt for å gjøre seriene stasjonære. De resulterende residuaene ble deretter simulert med de forskjellige modelltypene før de ble tilbaketransformert og validert.

Resultatene viste at SAR-LSTM nettverksmodellen ikke tilstrekkelig gjenskapte de statistiske kriteriene. VAR og VARMA modellene gjenskapte kriteriene på en tilfredstillende måte og ga dermed en god beskrivelse av H_s og T_p. Markovkjede modellen produserte bedre resultater enn SAR-LSTM modellen, men dårligere enn VAR og VARMA modellene. Konklusjonen på det første forskningsspørsmålet ble dermed at SAR-LSTM nettverksmodellen produserte resultater med en lavere kvalitet enn tradisjonelle stokastiske havgeneratorer. Videre viste resultatene at den lognormale Rosenblatt transformasjonen burde brukes til fordel for den oftere brukte Box-Cox transformasjonen hvis det er viktig å gjenskape fellesfordelingen.

SAR-LSTM modellen ble vist å være upraktisk siden det kreves mye tid og ressurser for å implementere en stokastisk generator. Konklusjonen på det andre forskningsspørsmålet blir dermed at dyp læring stokastiske havgeneratorer ikke er praktiske i simuleringsbasert design, med mindre kun et fåtall havgeneratorer er nødvendig. Det ble vist at de andre modelltypene var praktiske i simuleringsbasert design.

VAR og VARMA modellene var det beste modellvalget siden de gav høyest kvalitet på resultatene og var meget praktiske. Utvidelsen fra VAR til VARMA resulterte i bedre resultater på noen kriterier, men hadde den baksiden at det var mer krevende å estimere modell parameterne.

Contents

Problem Description i
Preface ii
Abstract iii
Sammendrag iv

1 Introduction **1**
1.1 Background 1
1.2 Research Questions 2
1.3 Structure of the Master's Thesis 3

2 Literature Review **4**
2.1 Generalities of Wind and Sea State Conditions 4
2.1.1 Data Sources 4
2.1.2 Environmental Parameters 5
2.2 Temporal Scales of Wind and Sea State Conditions 5
2.2.1 Short-term Wind and Sea State Conditions 5
2.2.2 Long-term Wind and Sea State Conditions 6
2.3 Stochastic Metocean Generator Requirements 7
2.3.1 Practicality 7
2.3.2 Quality of Results 7
2.4 Statistical Models vs. Physics-based Models 8
2.5 Stochastic Metocean Generators 9
2.5.1 Models Based on Gaussian Processes 9
2.5.2 Markov Chains 11
2.5.3 Parametric Models 12
2.5.4 Machine Learning 15
2.5.5 Combination Models 16
2.6 Model Validation Methods 17

3 Introduction to Deep Learning **18**
3.1 Learning Algorithms 18
3.1.1 Task 18
3.1.2 Performance 19
3.1.3 Experience 19
3.2 Goodness of Fit 19
3.2.1 Model Capacity 21

3.2.2 Bias and Variance in Learning . . . 22
3.2.3 Hyperparameters and Datasets . . . 23
3.3 Feedforward Neural Networks . . . 24
3.3.1 Feedforward Neural Network Architecture . . . 25
3.3.2 Cost Functions . . . 27
3.3.3 Output Unit Functions . . . 27
3.3.4 Hidden Unit Functions . . . 28
3.4 Optimization of Deep Learning Algorithms . . . 29
3.4.1 Batch and Minibatch algorithms . . . 29
3.4.2 Epochs and Iterations . . . 30
3.4.3 Stochastic Gradient Descent Algorithm . . . 30
3.4.4 Adam Algorithm . . . 31
3.4.5 Back-propagation Algorithm for Computing the Gradient . . . 32
3.4.6 Optimization Algorithm Selection . . . 32
3.5 Regularization . . . 33
3.5.1 Parameter Norm Penalties (L^1 and L^2) . . . 33
3.5.2 Dropout . . . 34
3.5.3 Early Stopping . . . 35
3.6 Recurrent Neural Networks . . . 35
3.6.1 Computational Graphs and Unfolding . . . 35
3.6.2 General Recurrent Neural Network . . . 37
3.6.3 The Gradient in Recurrent Neural Networks . . . 37
3.6.4 Long Short-term Memory (LSTM) Network . . . 38

4 Traditional Methods for Metocean Simulation 41
4.1 Vector Autoregressive Moving-average Models . . . 41
4.1.1 Autoregressive (AR) Model . . . 41
4.1.2 Moving-average (MA) Model . . . 41
4.1.3 Vector Autoregressive Moving-average (VARMA) model . . . 42
4.1.4 White Noise . . . 43
4.2 Discrete-Time, Finite-State Markov Chains . . . 43

5 Method 45
5.1 Description and Preprocessing of Observed Time Series . . . 45
5.1.1 North Sea Series . . . 45
5.1.2 Mediterranean Sea Series . . . 46
5.2 Data Transformations . . . 47
5.2.1 VAR, VARMA and SAR-LSTM Network Transformation . . . 47
5.2.2 Markov Chain Transformation . . . 50
5.3 Markov Chain Stochastic Generator . . . 50
5.4 VAR and VARMA Stochastic Generator . . . 52
5.5 SAR-LSTM Stochastic Generator . . . 54
5.5.1 Supervised Learning Problem . . . 54
5.5.2 LSTM Network Architecture and Functions . . . 55
5.5.3 Grid Search for Optimal Hyperparameters . . . 56

5.5.4 Transforming a Prediction Model to a Simulation Model . . . 57
5.6 Validation . . . 58

6 Results **59**
6.1 Grid Search Results . . . 60
6.2 Validation Results . . . 60
6.2.1 Lower Order Statistical Moments . . . 60
6.2.2 Joint and Marginal Distributions . . . 62
6.2.3 Auto- and Crosscorrelation . . . 63
6.2.4 Persistence . . . 64
6.3 Prediction and Loss Performance of LSTM Network Models . . . 65
6.4 Distributed Observed Scatter Plots . . . 66
6.5 Scatter Plots and Distributions of Residuals . . . 66
6.6 Curve Fit for Lognormal Conditional Parameters . . . 68

7 Discussion **69**
7.1 Metocean Series . . . 69
7.1.1 Hindcast Data: Building a Model on a Model . . . 69
7.1.2 Distribution of Observed T_p Values . . . 70
7.2 Transformations and Residuals . . . 71
7.2.1 Lognormal Assumption . . . 71
7.2.2 Stationarity of Residuals . . . 71
7.2.3 Covariance of Residuals . . . 72
7.2.4 Distribution of Residuals . . . 72
7.2.5 Curve Fit . . . 73
7.2.6 Rosenblatt Transformation Compared to Box-Cox Transformation . . . 73
7.3 Quality of Results . . . 74
7.3.1 Overall Lower Order Statistical Moments . . . 74
7.3.2 Monthly Lower Order Statistical Moments . . . 75
7.3.3 Joint and Marginal Distributions . . . 76
7.3.4 Auto- and Crosscorrelation . . . 77
7.3.5 Persistence of Significant Wave Height . . . 77
7.4 Practicality . . . 79
7.5 Ranking of Models . . . 80
7.6 Deep Learning in Simulation-based Design . . . 81
7.6.1 Training and Validation Error . . . 81
7.6.2 Stochastic Autoregressive LSTM Network Models . . . 81

8 Conclusion and Further Work **83**
8.1 Conclusion . . . 83
8.2 Further Work . . . 84

Bibliography **I**

Appendices **VIII**

A List of Acronyms **IX**

B Validation Results **XI**
B.1 North Sea Validation Results XI
B.2 Mediterranean Sea Validation Results XIX
B.3 Box-Cox Comparison Results for VARMA Model on North Sea Series XXVII

C Code **XXIX**
C.1 Support Functions XXIX
C.2 Residual Transformation Scripts XXXVII
C.3 Markov Model Scripts XLIII
C.4 VAR Model Script L
C.5 VARMA Model Script LIII
C.6 SAR-LSTM Network Model Scripts LIV
C.7 Validation Scripts LXIV

List of Figures

3.1 Idealized Examples of Machine Learning Fit (Goodfellow et al. (2016) Figure 5.5 Section 5.2) . . . 20
3.2 General Development of Training and Generalization Error as a Function of Capacity (Goodfellow et al. (2016) Figure 5.3 Section 5.2) . . . 21
3.3 General Development of Bias and Variance as a Function of Capacity (Goodfellow et al. (2016) Figure 5.6 Section 5.4.4) . . . 23
3.4 Graph Representation of Feedforward Network (Adapted from Nielsen (2015) Chapter 1) . . . 26
3.5 Unfolded Computational Graph and Circuit Diagram Representation (Goodfellow et al. (2016) Figure 10.2 Section 10.1) . . . 37
3.6 Circuit Diagram and Unfolded Computational Graph of General Recurrent Network (Goodfellow et al. (2016) Figure 10.3 Section 10.2) . . . 38
3.7 Block Diagram of LSTM Network Cell (Adapted from Goodfellow et al. (2016) Figure 10.16 Section 10.10.1) . . . 40

6.1 Validation Plot of Monthly Mean of H_s and T_p . . . 61
6.2 Validation Plot of Monthly Variance and Covariance of H_s and T_p . . . 61
6.3 Validation Scatter Plot of H_s and T_p for Mediterranean Sea Series . . . 62
6.4 Validation Scatter Plot of H_s and T_p for North Sea Series . . . 62
6.5 Validation Plot of H_s and T_p Autocorrelation Function . . . 63
6.6 Validation Plot of H_s and T_p Crosscorrelation Function . . . 63
6.7 Validation Plot of Normalized Persistence Contours Above Thresholds . . . 64
6.8 Validation Plot of Normalized Persistence Contours Below Thresholds . . . 64
6.9 Plot of LSTM Predictions on North Sea Test Set . . . 65
6.10 Plot of Training and Validation Loss for LSTM Models . . . 65
6.11 Scatter Plot of Distributed T_p and H_s for each Observed Series . . . 66
6.12 Scatter Plot of u_2 and u_1 for each Observed Series . . . 66
6.13 Histogram of u_1 for each Observed Series . . . 67
6.14 Histogram of u_2 for each Observed Series . . . 67
6.15 Curve Fit for Conditional Mean of $ln(T_p)$ Given H_s . . . 68
6.16 Curve Fit for Conditional Variance of $ln(T_p)$ Given H_s . . . 68

B.1 Comparison Plot of Monthly Mean for North Sea Series . . . XII
B.2 Comparison Plot of Monthly Variances and Covariance for North Sea Series . . . XIII
B.3 Comparison Plot of Joint and Marginal Distribution for North Sea Series . . . XIV

B.4 Comparison Plot of Autocorrelation Function for North Sea Series XV
B.5 Comparison Plot of Crosscorrelation Function for North Sea Series XVI
B.6 Comparison Plot of Normalized Persistence Above Threshold Contours for North Sea Series XVII
B.7 Comparison Plot of Normalized Persistence Above Threshold Contours for North Sea Series XVIII
B.8 Comparison Plot of Monthly Mean for Mediterranean Sea Series XX
B.9 Comparison Plot of Monthly Variances and Covariance for Mediterranean Sea SeriesXXI
B.10 Comparison Plot of Joint and Marginal Distribution for Mediterranean Sea Series . XXII
B.11 Comparison Plot of Autocorrelation Function for Mediterranean Sea Series XXIII
B.12 Comparison Plot of Crosscorrelation Function for Mediterranean Sea Series XXIV
B.13 Comparison Plot of Normalized Persistence Above Threshold Contours for Mediterranean Sea Series XXV
B.14 Comparison Plot of Normalized Persistence Below Threshold Contours for North Sea Series XXVI
B.15 Comparison Plots of VARMA models with Box-Cox and Lognormal Rosenblatt Transform - Part I XXVII
B.16 Comparison Plots of VARMA models with Box-Cox and Lognormal Rosenblatt Transform - Part II XXVIII

List of Tables

2.1 Comparison of Physics-based and Machine Learning Based Models (Erikstad (2017) Table 1) 8

5.1 Hyperparameters for Grid Search 57

6.1 Hyperparameters for LSTM Networks 60
6.2 Overall Lower Order Statistical Moments for North Sea Series 60
6.3 Overall Lower Order Statistical Moments for Mediterranean Sea Series 60
6.4 Root Square Mean Error (RMSE) for North Sea Test Set Predictions 65

B.1 Overall Lower Order Statistical Moments for North Sea Series XI
B.2 Overall Lower Order Statistical Moments for Mediterranean Sea Series XIX
B.3 Overall Lower Order Statistical Moments for Box-Cox and Rosenblatt Comparison XXVII

Chapter 1

Introduction

1.1 Background

The complexity of maritime supply chains is ever increasing due to a higher degree of integration with onshore logistics, consolidation creating larger fleets and stricter environmental regulations (Fathi et al., 2013). The complexity is further driven by new market requirements such as energy efficiency and increased flexibility (Gaspar, 2013).

This has led to a distinct trend within marine systems design to move away from traditional static design methods to more dynamic methods in order to understand value-driving aspects during early design stages (Fathi et al., 2013). Traditional design methods are unable to cope with complex operational profiles since these methods normally depend on simplified calculations based on some average or a few representative conditions (Hagen and Grimstad, 2010; Fathi et al., 2013). These methods fails in capturing the "complete" set of real-life operating conditions. The resulting hyper-optimization often lead to a suboptimal solution relative to a more simple and robust solution due to disruptions from neglecting real-world conditions (Fathi et al., 2013).

State-of-the-art design methods aims to understand the needs of the stakeholders and generate value-robust designs (Gaspar, 2013). Simulation-based design is one such state-of-the-art method which focuses on optimization over a wide range of conditions, thus creating a system that can deliver value over changing contexts (Fathi et al., 2013). Simulation-based design approaches generally consists of an operation model, which includes relevant elements such as ports, routes and mission-specific tasks (Fathi et al., 2013). The operation model is combined with metocean data to produce a realistic description of the operating conditions in the temporal and spatial domain (Fathi et al., 2013).

Thus, the availability and quality of metocean data becomes an integral part of simulation-based design. A tremendous amount of data on wind and sea state conditions are collected each day, but it remains relatively sparse to the size of the ocean space (Monbet et al., 2007). Hence, metocean simulation models are necessary to produce several realistic metocean scenarios for simulation-based design applications. Several metocean scenarios are especially important for applications which depends on repeated random sampling (Monte Carlo methods).

Metocean simulation is most appropriately viewed in the context of stochastic metocean generators. A stochastic generator is a statistical model whose realizations statistically match observed patterns (Hering et al., 2015). A stochastic generator in this context takes a time series as input and generates one or more synthetic time series of realizations.

Deep learning methods have dramatically improved the state-of-the-art in speech recognition, natural language processing, image recognition and many other domains (Krizhevsky et al., 2012; LeCun et al., 2015). Several studies have shown that deep learning models improve prediction performance for metocean data compared to traditional methods, but the potential of deep learning models as stochastic generators in simulation-based design currently remains unknown (Monbet et al., 2007).

1.2 Research Questions

The author was not able to find a single study on the use of deep learning models as stochastic generators in the context of simulation-based design. Additionally, there is a lack of literature on deep learning stochastic generators in general. Hence, there is a need for investigating the potential of deep learning models as stochastic metocean generators. Thus, the first research question becomes:

> *How does the quality of a deep learning stochastic metocean generator compare to simpler traditional stochastic metocean generators?*

Simulation-based design sets some practical restrictions for stochastic metocean generators. A vessel transits through vast distances where it experiences different metocean characteristics, dependent on the spatial and temporal domain. For instance, in deep-sea shipping applications the number of locations which a vessel transits with different metocean characteristics is significant. In simulation-based design, the voyage is generally discretized into several areas, where the ocean space in an area has approximately the same metocean conditions. The necessary fidelity of the discretization is dependent on the application.

Several stochastic generators, one for each area, must therefore be designed and implemented. This sets practical restrictions for the time and resources required for building a stochastic generator for a single area. Deep learning models require significant resources and time for training, which may lead them to being unpractical in a simulation-based design context.

A more complex and time-consuming model may produce results of a higher quality than a simpler model. This trade-off between practicality and quality of results must also be further understood. This leads to the second research question:

> *Can deep learning models be practically used as stochastic metocean generators in simulation-based design applications?*

1.3 Structure of the Master's Thesis

The structure of the master's thesis is as follows:

- **Literature Review**
  Chapter 2 consists of a literature review of the state-of-the-art within stochastic metocean simulation. The chapter begins with a description of relevant data sources, definition of environmental parameters and the concepts of short-term and long-term wind and sea state conditions. Literature on metocean simulation is then presented with emphasis on methods which are most relevant for simulation-based design applications.

- **Theory**
  Chapter 3 gives an introduction to deep learning with a focus on deep neural networks for sequence processing. Chapter 4 details the theory of two commonly used methods for metocean simulation: Discrete time finite-state Markov chains, vector autoregressive processes (VAR) and vector autoregressive moving-average processes (VARMA).

- **Method**
  Chapter 5 presents the methods used for implementation and validation of the stochastic metocean generators. First, two hindcast time series to be used as case study for validation is presented. Then, the transformations applied before and after simulation are introduced. Finally, the simulation method for each of the stochastic generators and the quality of results validation criteria is presented.

- **Results and Discussion**
  Chapter 6 presents the results of the case study given in Chapter 5. The results and methods used are then discussed, analyzed and validated in Chapter 7.

- **Conclusion and Further Work**
  Chapter 8 concludes on the research questions posed in Chapter 1 and gives recommendations for further work to be done within this topic.

Chapter 2

Literature Review

This chapter sets the stage for the remainder of the thesis by first presenting an introduction to the most common metocean conditions for simulation, wind and sea state conditions. Relevant data sources, definition of environmental parameters and the concepts of short-term and long-term wind and sea state conditions is then presented. This is followed by general requirements for a stochastic metocean generator and a literature review on methods for wind and sea state simulation, with emphasis on methods most relevant for simulation-based design applications. Finally, methods for quality of results validation are presented.

2.1 Generalities of Wind and Sea State Conditions

2.1.1 Data Sources

Monbet et al. (2007) classifies data sources which detail wind and sea state conditions as:

- In-situ observations measured by e.g., buoys, ships and satellites
- Hindcast, nowcast or forecast data from meterological models

Woolf and Challenor (2002), Caires and Sterl (2005) and Izquierdo and Guedes (2005) compared the quality of the aforementioned types of data sources. Buoy data series are seldom longer than 10 years and are often subject to missing and noisy data. The sampling frequency can vary from 10 minutes to 10 hours. Satellite data are sampled at a much higher frequency along the satellites path, but the data generated for a specific location is sparse.

Data from numerical meteorological models are generally easier to use since the length of the time series are much longer, up to 50 years, and generally contains no missing values. The main drawback of numerical models is that they are often smoother than real data and tends to underestimate the occurrence of extreme events (Woolf and Challenor, 2002; Caires and Sterl, 2005).

2.1.2 Environmental Parameters

There is a significant amount of environmental parameters with several possible definitions that can be used to describe wind and sea state conditions. Below follows an introduction to the most important environmental parameters and their definitions as given by Det Norske Veritas (2010):

- Significant wave height H_s [m]: Average height (trough to crest) of the highest one-third waves in the indicated time period.
- Peak period T_p [s]: Wave period determined by the inverse of the frequency at which a wave energy spectrum has its maximum value.
- Zero-up-crossing period T_z [s]: Average time interval between two successive up-crossings of the mean sea level.
- Mean wave direction θ_m [deg]: Direction which the waves are coming from (0/360 degrees is from north, 180 degrees is from south).
- Wave peak direction θ_p [deg]: Direction which the wave peaks are coming from.
- Wind intensity U [m/s]: Mean wind speed at an elevation of 10 m over a fixed period (10 minutes to 3 hours in general).
- Mean wind direction ϕ [deg]: Mean direction from which the wind is blowing.

2.2 Temporal Scales of Wind and Sea State Conditions

2.2.1 Short-term Wind and Sea State Conditions

Spellman (2016) defines a sea state as the general condition of the free surface on a large body of water, with respect to wind waves and swells, at a specific location and moment. Det Norske Veritas (2010) states that the sea state is determined by a wave frequency spectrum, with a given significant wave height H_s, a representative frequency T, a mean propagation direction and a spreading function.

It is normally assumed that a sea state is a stationary stochastic process (Det Norske Veritas, 2010). A common assumption in relation to short-term wave conditions is that the sea surface is stationary for a duration of 20 minutes to 3-6 hours, but the period of stationary can vary from 30 minutes to 10 hours (Det Norske Veritas, 2010). Three hours has been introduced as the standard time between each sample when measuring sea states (Det Norske Veritas, 2010). Det Norske Veritas (2010) further states that the wave conditions during a sea state can be divided into two classes: wind seas and swells. Local winds generate the wind seas while the swells have no relationship to the local wind. Swells are waves which have traveled out of the area where they were generated and several components may be present at a specific location.

A wave spectrum is the power spectral density function of the vertical sea surface displacement (Det Norske Veritas, 2010). These are either given in a table, measured spectra or by an analytic parameter function. The selection of appropriate wave spectra is dependent on the geographical area (Torsethaugen, 1996). Several standardized spectra are available and commonly used in the maritime industry. The JONSWAP spectrum and Pierson-Moskowitz (PM) spectrum are often applied when analyzing wind seas (Hasselmann et al., 1973; Pierson and Moskowitz, 1964). The Torsethaugen spectrum is two-peaked, meaning that it can be used to account for both wind sea and swells, which is often the case in low to moderate sea states in open areas (Torsethaugen, 1996).

The wind climate is also assumed to be stationary in the short-term, normally a shorter period than for waves, often assumed to be 10 minutes (Det Norske Veritas, 2010). The short-term mean representation is not intended to cover extreme conditions experienced in tropical regions such as hurricanes, cyclones or typhoons. Neither is it intended for small scale extreme events such as wind gusts, as these are transient in both speed and direction (Det Norske Veritas, 2010).

2.2.2 Long-term Wind and Sea State Conditions

The long-term variation of wave climate can be described in terms of generic distributions, or in terms of scatter diagrams for governing sea state parameters (Det Norske Veritas, 2010). A scatter diagram provides the frequency of occurrence for a given parameter pair, such as H_s and T_p. Wave climate can be described by both marginal distributions and joint environmental models. The generic models are generally established by fitting distributions to wave data from the actual area (Det Norske Veritas, 2010).

Det Norske Veritas (2010) states that long-term analysis is often dived into two different approaches, global and event models. Global models are based on all available data from a long-term time series of observations in the given area, while event models looks at observations which are above some threshold.

Before modeling any process, it is necessary to understand the underlying statistical structure of the data series to recreate their fundamental qualitative and quantitative characteristics. Athanassoulis and Stefanakos (1995) states that for any long-term spectral wave parameter, and especially for times series of significant wave height H_s, the prevailing qualitative features are

1. Fluctuating character of the sequence of observations that prevents any exact reproducibility of the time series and calls for stochastic modeling.
2. Statistical dependency between observations separated by a relativly small time period.
3. Pronounced seasonal variability within the annual cycle.
4. Yearly statistical periodicity induced by the meteorological annual cycle.
5. An over-year trend which may be difficult to identify.

Sampling Frequency

Studying sea state time series requires that the sampling frequency must not be greater than the mean value of the stationary duration of the sea states (Det Norske Veritas, 2010). Hence, the sampling frequency should be approximately 3-6 hours to be considered satisfactory (Athanassoulis and Stefanakos, 1995). The frequency for wind series should be less than that of sea states, 10 minutes to 1 hour, since they have a shorter stationary duration (Det Norske Veritas, 2010).

2.3 Stochastic Metocean Generator Requirements

Stochastic generators are built for a specific purpose and the definition of their goodness is dependent on the process under consideration and the end-user of the model (Monbet et al., 2007). When concerned with wind and sea state conditions, the primary uses of the models are forecasting, hindcasting, simulation and reconstruction of missing values. Monbet et al. (2007) states that the goodness of a model relies on the trade-off between its practicality and its accuracy in describing various features of the physical phenomenon, i.e., quality of results.

2.3.1 Practicality

The practicality can be assessed from a myriad of criteria. First, the model should be robust against missing and errorous observations from the data source (Monbet et al., 2007). The data may be observed at a very low frequency or only descriptive statistics may be available (Vik, 1981; Hogben and Standing, 1987).

Monbet et al. (2007) further states that the generator must be robust against the very nature of the process it is modeling. Processes have their own specific characteristics such as spatial and temporal dependencies. If the process is multivariate then it is important that the model can recreate dependencies between the components.

There are also numerous mathematical and computational properties which must also be considered such as the required amount and fidelity of data, number of parameters required for model type and amount of time needed for implementation and running (Monbet et al., 2007).

2.3.2 Quality of Results

Comparison statistics between the computed model and the observed time series can be used to evaluate the quality of results. Several statistics and validation methods are available, but often only graphical comparison is performed (Monbet et al., 2007). Section 2.6 presents several methods for determining the quality of results.

2.4 Statistical Models vs. Physics-based Models

An alternative to the statistical models presented in this chapter are physics-based models for simulation. Erikstad (2017) states that physics-based models are models which are built on engineering analysis with foundations in Newtonian mechanics. Erikstad (2017) further states that machine learning models are based on very different principles than physics-based models. Table 2.1 is adapted from Erikstad (2017) and shows a comparison between physics-based and machine learning based models in the context of digital twins.

Table 2.1: Comparison of Physics-based and Machine Learning Based Models (Erikstad (2017) Table 1)

	Machine Learning	**Physics**
Pro	• Model derived from data only - no need for domain knowledge • Generic and flexible - handles heterogeneous data streams (also non-physics) • Good at discovering complex relations and patterns	• Models capture deep existing knowledge based on Newtonian physic • Causal relationships provide insight and understanding • Uncertainty controlled by input and modeling accuracy • Model has universal validity - predict any point covered by model
Con	• The availability of training data needed to develop model • Approximation methods, no exact mathematics • Correlations, not causality. Black-box, no explanations (in particular, deep learning) • Predictive capabilities deteriorate quickly outside training set scope • Difficult to predict extreme/critical conditions (few observations)	• Require extensive domain (physics) knowledge • Computationally intensive, challenge in real-time • Complete assumptions about input-output must be made upfront

Much of what is presented in Table 2.1 also holds for statistical models in general. Statistical models are generally derived from data only and have no requirements for domain knowledge. Statistical models are further generic, flexible and good at discovering complex representations, dependent on complexity of the model.

The drawback of statistical models is that they are often approximation methods which are limited by the availability of data and are based on correlations, not causality. Some statistical models also struggle with predicting extreme events due to the low number of observations of these events.

2.5 Stochastic Metocean Generators

The following section presents methods for metocean simulation. Emphasis has been placed on the models which are most relevant for simulation-based design applications. The models are classified according to the system set forth by Monbet et al. (2007).

2.5.1 Models Based on Gaussian Processes

The history of non-stationary modeling of wave time series start with the work of O'Carroll (1984) which presented ideas for univariate and bivariate modeling of wave heights. It was not until the early 90s that literature concerning time series analysis of sea state parameters first appeared with a focus on developing wave simulators (Medina et al., 1991; Bettencourt, 1993; Borgman and Scheftne, 1991; Walton and Borgman, 1990). Ocean and marine engineers had used times series data prior to this, but with a traditional focus on specific statistical events and drew inference from these using simplified models which did not take into account the non-stationarity and correlation between the parameters (Athanassoulis and Stefanakos, 1995).

Lognormal Models

Medina et al. (1991) measured significant wave height H_s and zero up-crossing period T_z at the coast of Oregon which they used to simulate and study bivariate time series of the same parameters. They assumed that both parameters followed a lognormal distribution. A seasonal standardization transformation was applied

$$x(i) = \frac{h_s(i) - A_h(i)}{B_h(i)} \tag{2.1}$$

$$x(i) = \frac{t_z(i) - A_t(i)}{B_t(i)} \tag{2.2}$$

where $h_s = log(H_s)$, $t_z = log(T_z)$. $A_{h,t}(i)$ and $B_{h,t}(i)$ are annually periodic parameters. The periodic parameters were estimated on a month-by-month basis and were required to account for seasonal variability. This transforms the time series into a normalized stationary Gaussian time series, $\{(x(i), y(i)), i = 1,2,\ldots\}$. The series was then simulated with a first-order autoregressive AR(1) model with constant coefficients. Autoregressive and moving-average models will be detailed in Section 4.1.

Bettencourt (1993) continued the work of Medina et al. (1991) by first showing that significant wave height H_s, spectral peak period f_p and spectral moments m_1 and m_2 also follow a lognormal distribution. Bettencourt (1993) then constructed models based the Medina et al. (1991) approach for the pairs (H_s,m_1), (H_s,m_2) and (H_s, f_p). When validating the results, various inadequacies were reported which warranted further developments in modeling and estimation techniques. The goal of Bettencourt (1993) was not to study the sea state time series structure in detail, but to create fast and practical sea state simulators.

Normal Scores

Borgman and Scheftne (1991) studied a trivariate 20-year hindcast time series of H_s, T_p and θ_m. Instead of assuming a lognormal distribution, Borgman and Scheftne (1991) used a normal-scores transformation and assumed piece-wise (month-by-month) stationarity of the series. The data was transformed to a normal sequence and the mean monthly correlation structure was calculated. This allowed for preserving the primary statistical properties of the dataset and create a normal, piece-wise stationary model for simulating the trivariate series.

Walton and Borgman (1990) developed a procedure for simulating the water levels of the Great Lakes, where the water level was assumed to be an univariate non-Gaussian and seasonally non-stationary time series. The time series of seasonal trends for mean and standard deviation was found by applying low-pass filtering. The time series was then seasonally standardized, by same method as shown in Equation 2.1 and 2.2, and assumed stationary. The use of the empirical distribution function combined with additional analytical representations for the tails allowed for the normal score transformation to create a Gaussian time series.

Non-stationary Stochastic Processes

Athanassoulis and Stefanakos (1995) built on the work of the aforementioned and introduced a method for modeling univariate long-term wind and wave data time series as a non-stationary stochastic process with yearly periodic mean and standard deviation. As previously discussed, Athanassoulis and Stefanakos (1995) showed that seasonal variability in long-term wind and sea state time series becomes apparent within the annual cycle. Further, a non-negligible year-to-year variability was also seen when comparing mean annual values. Athanassoulis and Stefanakos (1995) showed that a univariate time series of wind and sea state condition can be represented by the decomposition

$$Y_t = m_t + s_t W_t \tag{2.3}$$

where W_t is a zero-mean stationary (in general, non-Gaussian) stochastic process. Additionally, s_t is assumed periodic with a period of one year, while m_t could contain both periodic and non-periodic terms. Methods for determining m_t and s_t was also presented.

Athanassoulis and Stefanakos (1995) further showed that the stochastic process Y_t is not periodic, but periodically correlated (cyclostationary), meaning the associated probability distributions vary periodically with time. A stochastic process Z_t is said to be cyclostationary if and only if Z_t can be represented on the form

$$Z_t = \sum_{k=-\infty}^{+\infty} W_{k,t} e^{ikw_0 t} \tag{2.4}$$

where the processes $W_{k,t}$ are all stationary and stationary correlated (Athanassoulis and Stefanakos, 1995). Cyclostationary processes are widely applicable for modeling several environmental phenomenons and the underlying mathematical theory is well-developed and advantageous for simulation and extreme-value prediction (Konstant and Piterbar, 1993).

Stefanakos and Belibassakis (2005) extended the work done on univariate non-stationary stochastic models in Athanassoulis and Stefanakos (1995) to the multivariate domain, by developing a non-stationary stochastic model, suitable for the analysis and simulation of multivariate time series of wind and sea states. The result was a model belonging to the class of periodically correlated stochastic processes with yearly periodic mean value and standard deviation (cyclostationary stochastic process).

First, an appropriate Box-Cox transformation was applied to the wind and sea state data, which transforms the data into an approximately Gaussian series. The univariate decomposition of a long-term time series, as shown in 2.3, can be extended to the multivariate case

$$\boldsymbol{Y}_t = \boldsymbol{M}_t + \boldsymbol{\Sigma}_t \boldsymbol{W}_t \tag{2.5}$$

where $\boldsymbol{M}_t$ is a $N \times 1$ vector and $\boldsymbol{\Sigma}_t$ is an $N \times N$ matrix. Both are deterministic periodic functions with period of one-year and describe the seasonal patterns in the data. In the multivariate case, the seasonal deviation s_t has been replaced by $\boldsymbol{\Sigma}_t$ which is the square root of the covariance matrix. $\boldsymbol{W}_t$ is a $N \times 1$ zero-mean, stationary stochastic process vector.

Stefanakos and Athanassoulis (2003) previously showed that the residual component $\boldsymbol{W}_t$ can be considered stationary. This gives $\boldsymbol{W}_t$ the structure of a periodically correlated stochastic process. Finally, due to the Box-Cox transform of the original series, $\boldsymbol{W}_t$ is Gaussian stationary and a suitable multivariate parametric model can then be fitted.

Stefanakos and Belibassakis (2005) fitted a vector autoregressive moving average (VARMA) model to the stationary Gaussian process. This method allows for effective reproduction of available information about wind and sea state conditions. The output from the VARMA model was shown to be of good quality and enables calculation of all common wind and sea state statistics.

2.5.2 Markov Chains

Hagen et al. (2013) presented a multivariate Markov chain model for generating sea state time series based on observed time series. Stationarity is a requirement for a Markov chain model and Hagen et al. (2013) presented two distinct models to deal with the seasonality of the observed time series.

The first model was based on assuming piece-wise (month-by-month) stationary of the series and then generating a transition matrix for each month. This approach for handling seasonality has been frequently used in previous studies (Scheu et al., 2012a,b).

The second model followed the work of Jim and Chou (2002) and normalized the observed time series by

$$y^*_{t,p} = \frac{y_{t,p} - \bar{y}_p}{S_p}, \quad p = 1, 2, \ldots, 12 \tag{2.6}$$

where $y_{t,p}$ is an observed sea state parameter at time t in month p. $\bar{y}_p$ and S_p are the mean and standard deviation for the observed values in the month p. The resulting transformed time

series was assumed to be stationary due to that the seasonal variation in the mean and standard deviation are the dominant components of the time series non-stationarity (Hagen et al., 2013). The transition probabilities for the multivariate sea state states were estimated from observed data, and each sea state parameter was discretized by dividing their range into equally sized bins. The inverse transform was applied after simulation and both models were able to reproduce the lower order statistical moments and persistence.

2.5.3 Parametric Models

Time-varying Autoregressive Models

Huang and Chalabi (1995) proposed a linear, time-varying autoregressive process to forecast wind speed which accounted for the non-stationary nature of wind speed. The time-varying parameters are modeled by smoothed, integrated random walk processes and estimated by a Kalman filter. The time-varying autoregressive model of order r is given by

$$U_t = \sum_{i=1}^{r} a_{t,i} U_{t-i} + \epsilon_t \tag{2.7}$$

where Ω is a zero-mean white noise vector and Γ is a dummy vector of parameters which are either white noise or random walk processes. Let $\Psi = (a_{t,1}, \ldots, a_{t,r})$, then $\Psi_t = \psi_{t-1} + \Gamma_{t-1}$ and $\Gamma_t = H\Gamma_{t-1} + \Omega_t$. Ψ is an unknown parameter vector which is a type of random walk dependent on the diagonal matrix H.

Scotto and Guedes Soares (2000) proposed a self-exciting threshold autoregressive (SETAR) model to overcome some of the weaknesses of the linear models. Linear models have a symmetric joint distribution which leads to stationary Gaussian ARMA models being unsuitable to model data exhibiting strong asymmetry (Monbet et al., 2007). Long-term time series of significant wave height has been shown to be described by lognormal, Weibull or other asymmetrical distributions (Ferreira and Guedes Soares, 1999).

Scotto and Guedes Soares (2000) states that a time series follows a TAR model with threshold variable X_{t-d} if it satisfies:

$$X_t = \phi_0^k + \sum_{i-1}^{p} \phi_i^k X_{t-1} + \epsilon_t^k \quad r_{k-1} \leq X_{t-d} \leq r_k \tag{2.8}$$

where $k = 1, \ldots, g$ and d are positive integers. ϵ_t^k are a sequence of independent, identically distributed normal random variables with zero-mean and variance σ_k^2. The real numbers r_i satisfy $-\infty = r_0 < r_1 \ldots < r_g = \infty$.

Scotto and Guedes Soares (2000) applied the transformation $Y_t = ln(X_t)$ to fit the SETAR model to observed significant wave heights.

Thus the model becomes

$$Y_t = \sum_{i=1}^{r} a_i^{(S_t)} Y_{t-i} + b^{(S_t)} + \sigma^{(S_t)} \epsilon_t \tag{2.9}$$

with $S_t = i$ if and only if $Y_{t-d} \in [r_i, r_{i+1}]$ for a fixed integer d. Here, $r_1 < r_2 \ldots < r_M$ are parameters of the model and ϵ_t is Gaussian white noise.

The SETAR model was evaluated against a linear autoregressive model, AR(22). Scotto and Guedes Soares (2000) showed that the difference in the lower order statistical moments where negligible and this also applied to the autocorrelation function. For higher order moments, the non-linear model provides a better approximation of the skewness and kurtosis of the original data then the linear model.

Hidden Markov-switching Autoregressive Models

Ailliot and Monbet (2012) proposed the use of Markov-switching autoregressive (MS-AR) models to describe wind time series. The MS-AR model was initially proposed in (Hamilton, 1989) to describe economic time series and is a generalization of hidden Markov-switching (HMM) models and autoregressive models. Several autoregressive models are used to describe the evolution of the processes and the transition between the different AR models is controlled by a hidden Markov chain (Ailliot and Monbet, 2012).

Ailliot and Monbet (2012) showed that MS-AR models have the ability to model diverse time scales and improve the description of important dynamic properties such as the persistence of weather states. The MS-AR model was able to match statistical properties of the underlying data such as marginal distributions, second order moments and persistence. The main drawbacks was that the model failed in reproducing the lowest part of the marginal distribution and the inner-annual variability.

Hering et al. (2015) built on the aforementioned MS-AR model and introduced a Markov-switching vector autoregressive (MS-VAR) model for wind vector simulation. The model was able to simulate wind vectors at multiple locations simultaneously. In order to capture the seasonality and diurnal cycles (any pattern that recurs daily) the algorithm was generalized and a nonparametric transformation of the components to normality captured higher order statistical moments. The MS-VAR model showed promise in simultaneous simulation at multiple locations while maintaining the proper spatial relationships.

GARCH

Toll (1997) introduced the use of generalized autoregressive conditional heteroskedasticity (GARCH) models for wind speed. Random variables exhibits heteroscedastic if there are sub-populations that have different variabilities from others (Toll, 1997). The variability is in this case quantified by the variance. GARCH models describe the variance of the current error term as a function of the previous error terms. (Toll, 1997) let the conditional variance of an observation depend

linearly on the conditional variances of the previous observations and prediction errors.

Toll (1997) assumed that the wind speed U was Markovian of order r and the conditional distribution of Y_t given $(Y_{t-1}, \ldots, Y_{t-r})$ is described by a gamma distribution with mean

$$\mu_t = \sum_{i=1}^{r} a_i Y_{t-i} + b \tag{2.10}$$

and variance

$$\sigma_t^2 = \alpha + \sum_{i=1}^{p} \lambda_i (Y_{t-i} - \mu_{t-i})^2 + \sum_{i=1}^{q} k_i \sigma_{t-i}^2 \tag{2.11}$$

The model in Toll (1997) was shown to outperform the homoscedastic alternative.

Copula-Based Models

Vanem (2016) studied various bivariate modeling techniques for the joint distribution of significant wave height and zero-crossing wave period. The models in the study consisted of a conditional lognormal model and several meta-models based on parametric copulas.

Vanem (2016) defines a copula as a multivariate probability distribution, whose variables have uniform marginals. More formally, Vanem (2016) states that the copula of two or more variables (X, Y) is the joint cumulative function $(F_X(x), F_Y(y))$. Vanem (2016) defines the copula as

$$C(u, v) = P[F_X(x) \leq u, F_Y(y) \leq v] \tag{2.12}$$

with a corresponding copula density $c(u, v)$, which is defined as the derivative of C with respect to each of its arguments. As a consequence, the joint probability density can be written as

$$h(x, y) = f(x)g(y)c(F(x), G(y)) \tag{2.13}$$

Vanem (2016) further showed that the copula models failed to capture the dependencies in the data and that the conditional model performed better. However, Vanem (2016) states that significant improvements could be made with more advanced copula construction techniques.

Leontaris et al. (2016) presented an alternative copula-based method for simulation of wind speed and significant wave height. The copulas were constructed to take into account the autocorrelation and dependence between the metocean parameters. The method was evaluated using a realistic cable installation scenario in addition to observed wind and sea state time series from the installation area. Leontaris et al. (2016) showed that the simulated time series from the copula model provided better insights in the operation compared to when only observed time series was used. The added insight arose from the ability to take more possible realizations into account. Additionally, the method adequately replicated the persistence, seasonality and workability in the observed time series.

2.5.4 Machine Learning

The most prevalent form of machine learning methods in metocean simulation and forecasting are artificial neural networks (Monbet et al., 2007), see (Goodfellow et al., 2016) for more information on ANN and deep learning in general. The first use of neural networks the author could find for wind and sea state parameters was a model comparison study for short-term forecasting of wind speed (Stephos, 2000). Several different neural networks were compared to an ARMA model and Stephos (2000) found that the neural networks performed better in validation and had several advantages over the traditional forecasting methods such as adaptability and error tolerance.

More and Deo (2003) presented another technique to forecast wind speed on daily to monthly temporal scales. The result showed that the neural networks were more accurate than traditional forecasting methods, including a higher correlation and lower deviations with observed data. Makarynskyy et al. (2004) proposed the use of neural networks to forecast significant wave heights and zero-up-crossing wave periods. Tsai et al. (2002) used a neural network for prediction of H_s given the highest one-tenth wave height $H_{1/10}$, the highest wave height H_{max} and the mean wave height H_{mean}. The results showed a satisfactory accuracy for wave prediction.

Bazargan et al. (2007) proposed a deep neural network for prediction of the mean zero-up-crossing wave period T_z for 3-hourly sea states. Several networks were trained by simulated annealing and the output was used to estimate parameters of a new conditional distribution for T_z, given mean zero-up-crossing wave periods and significant wave heights. The distribution was then used for forecasting. Bazargan et al. (2007) concluded that neural networks were effective for estimating the conditional distribution and that the model performed well for forecasting values of T_z in the near future.

Several other authors have also studied the use of neural networks for wind speed forecasting and a commonality is that neural network models generally obtain better short-term forecasts than those obtained by linear autoregressive moving-average models (Monfared et al., 2009; Cadenas and Rivera, 2009; Li and Jing, 2010).

In the last few years, the focus within machine learning has shifted towards the use of more complex neural networks, recurrent neural networks (RNN), for prediction of wind and sea states. Recurrent neural networks are generally better at processing sequential data than feedforward neural networks (Goodfellow et al., 2016).

Balluff et al. (2015) investigated the use of recurrent neural networks for forecasting short-term wind speed and pressure. The authors showed that this was indeed possible and the RNN model produced satisfactory results. Balluff et al. (2015) also concluded that a long short-term memory network (LSTM) model, a model in the RNN family, could further improve the results.

Xiaoyun et al. (2016) introduced the use of a deep long short-term memory (LSTM) model for prediction of wind power. The results showed that the LSTM model had a higher prediction accuracy and greater potential for engineering applications, compared to simpler machine learn-

ing methods such as feedforward networks and support vector machines (SVM). Lopez et al. (2016) used two recurrent networks models, LSTM and echo state networks, for wind speed forecasting. The results supported the findings of Xiaoyun et al. (2016) as the LSTM network model outperformed feedforward networks in forecasting several steps ahead.

The most relevant studies in the context of stochastic metocean generators by Aminzadeh-Gohari et al. (2008) and Toarmina et al. (2012). Aminzadeh-Gohari et al. (2008) built on the work done in Bazargan et al. (2007) in order to simulate the univariate series of H_s using seven deep feedforward neural networks. The neural networks were trained using simulated annealing and then used to estimate the parameters of a conditional probability density distribution for H_s, given preceding values of H_s. Aminzadeh-Gohari et al. (2008) states that the probability distribution for H_s at time step t was given by its eight past 3-hourly H_s values

$$f_{H_s|H_s(t_{i-1})} = h_{i-1}, \ldots, H_s(t_{i-8}) \tag{2.14}$$

where t_i the day of the i^{th} 3-hourly H_s. The random variable $H_s(t_i)$ denotes the significant wave height at t_i and h_i is the observed value for $H_s(t_i)$.

The seven neural networks were used to determine a hepta-parameter spline distribution which approximated the probability distribution of H_s. This probability density function was then used for simulation. Aminzadeh-Gohari et al. (2008) concluded that feedforward neural networks were suitable for determining the parameters of the hepta-spline distribution and that the resulting simulation of H_s performed well when validated in an extreme value analysis context.

Toarmina et al. (2012) presented a method for long-term simulation of groundwater levels with feedforward networks. A feedforward network was first trained to predict 1-hour ahead groundwater levels using past observed groundwater levels, rainfall and evapotranspiration. Then the simulation was carried out by iteratively feeding back predicted groundwater levels, in addition to real external data; rainfall and evapotranspiration. The feedforward network was able to accurately reproduce groundwater levels for several months.

The majors drawbacks of neural networks are the large number of parameters they involve and their lack of interpretability (Monbet et al., 2007). Further, training a neural network requires significant time and resources (Goodfellow et al., 2016).

2.5.5 Combination Models

Guanche et al. (2013) presented an ambitious methodology for simulating hourly sea state time series by combining several different techniques such as univariate ARMAs, autoregressive logistic regression and K-means clustering algorithms.

The methodology consists of three steps. First, Guanche et al. (2013) decomposed synthetic daily sea level pressure fields in to its principal components and simulated these using a multivariate technique first introduced by Morales et al. (2010). Then, a K-means clustering algorithm, as proposed by Camus et al. (2011), was applied to the daily mean sea conditions and

simulated by an autoregressive logistics regression model with previously simulated daily sea level pressure fields as covariates. Finally, the hourly sea states were conditioned on the previously generated daily mean sea condition patterns and simulated. The methodology proved very successful in reproducing the multivariate sea state time series of dependent variables. Guanche et al. (2013) states that the methodology could be extended to simulate multiple location simultaneously and incorporate climate change issues.

2.6 Model Validation Methods

The stochastic metocean simulators previously detailed in this chapter uses several different validation methods dependent on the application of the synthetic time series. Monbet et al. (2007) states that the validation method depends on the features of interest for the end user and many different characteristics of the synthetic series could be investigated .

Monbet et al. (2007) further states that the most common validation methods consists of criteria based on comparing specific statistics calculated from the observations with those corresponding to the realizations of the stochastic generator. There exists a large amount of possible statistics, such as statistical moments, distributions and dependencies.

In many applications there exists some temporal dependence, and in these applications the persistence below (or above) some given threshold becomes of great interest (Monbet et al., 2007). The persistence of weather windows and the waiting times between these weather windows is an importation property for many applications (Hagen et al., 2013). The persistence of weather windows can be defined as the amount of hours that the environmental parameters do not exceed some environmental thresholds (Leontaris et al., 2016). Additionally, the temporal variations in the statistical properties as a results of seasonal variations is also of interest for many applications.

Chapter 3

Introduction to Deep Learning

This chapter gives an introduction to the theory of deep learning and starts with an introduction to essential machine learning concepts. The quintessential deep learning model deep feedforward networks (FNN) is presented to provide an understanding of deep learning algorithms. The theory is then extended to recurrent neural networks (RNN) with emphasis on long short-term memory (LSTM) networks.

It should be noted that this chapter on deep learning draws significantly from the excellent book on deep learning by Goodfellow et al. (2016). The general structure of this chapter and the mathematical notation is based on Chapter 5-8 and 10-11 in Goodfellow et al. (2016).

3.1 Learning Algorithms

This section is based on Chapter 5 in Goodfellow et al. (2016) and presents fundamental learning theory which is necessary for understanding the deep learning algorithms presented later.

Mitchell (1997) defines learning in the context machine learning as:

> *"A computer program is said to learn from experience* $\boldsymbol{E}$ *with respect to some class of tasks* $\boldsymbol{T}$ *and performance measure* $\boldsymbol{P}$*, if its performance at tasks in* $\boldsymbol{T}$*, as measured by* $\boldsymbol{P}$*, improves with experience* $\boldsymbol{E}$*."*

3.1.1 Task

Goodfellow et al. (2016) states that tasks (**T**) are normally determined by how the algorithm processes an *example*, where an example is a set of quantitatively measured *features* from some event or object. More formally, a feature $\boldsymbol{x}_i$ is one entry in an example vector $\boldsymbol{x} \in \mathbb{R}^n$ (Goodfellow et al., 2016).

The range of machine learning tasks is quite diverse and Goodfellow et al. (2016) provides some examples of machine learning tasks including, but not limited to:

- Classification: Determine which category or class some input belongs in
- Regression or Prediction: Predict a numerical value based on some input
- Clustering: Grouping inputs into clusters such that the members of a cluster are more similar to each other than those of another cluster
- Anomaly detection: Detect atypical events or objects

3.1.2 Performance

Goodfellow et al. (2016) defines performance (**P**) as a quantitative measure which must be specified to evaluate the goodness of the algorithm and is usually determined by the task. For a classification task, the performance could be the accuracy of the model, i.e., the percentage of examples which are classified correctly (Goodfellow et al., 2016). For a prediction algorithm, the performance measure could be the amount of deviation or error (Nielsen, 2015). The performance is in most cases evaluated on a set of data which is distinct from the set used for training the algorithm (Goodfellow et al., 2016).

3.1.3 Experience

Finally, Goodfellow et al. (2016) defines experience (**E**) as the experience the algorithm gets during the learning process and can mainly be classified into either supervised or unsupervised. Supervised learning is the most common form of machine learning and also most prevalent within deep learning (LeCun et al., 2015).

Goodfellow et al. (2016) makes a separation of the terms by defining supervised learning as when the experience for the algorithm is a dataset of features where each example in the dataset is accompanied by a corresponding *target* or *label*. The supervised experience comes from training on examples $\boldsymbol{x}$ with a label $\boldsymbol{y}$ with the goal of learning to predict $\boldsymbol{y}$ from $\boldsymbol{x}$, usually from the distribution $p(\boldsymbol{y}|\boldsymbol{x})$ (Goodfellow et al., 2016).

Goodfellow et al. (2016) defines unsupervised learning as when the algorithm experiences examples without labels. The unsupervised experience consist of training on examples of $\boldsymbol{x}$ with the goal of learning $p(\boldsymbol{x})$ or some property of that distribution (Goodfellow et al., 2016).

3.2 Goodness of Fit

This section introduces several basic machine learning concepts of high importance for most deep learning algorithms. The concepts in Goodfellow et al. (2016) Chapter 5 most relevant for understanding deep learning algorithms are presented in this section.

A machine learning algorithm must be able to perform well not only on the data which it was trained, but on new unseen data. Goodfellow et al. (2016) defines *generalization* as the ability of the algorithm to perform well on unseen data.

The available dataset is usually split into a *training set* and a *test set* (Goodfellow et al., 2016). The training set is used to train the model and from this an error measure called the *training error* can be computed. Goodfellow et al. (2016) defines the *generalization error* or test error as the expected error of a previously unseen input computed from the test set. A machine learning algorithm aims to have both a low training and generalization error (Goodfellow et al., 2016).

Goodfellow et al. (2016) further defines the *data-generating process* as the probability distribution over the dataset that generates the training and test data. Goodfellow et al. (2016) assumes that the examples in each dataset are independent from each other. Further, Goodfellow et al. (2016) assumes that the training set and test set have the same probability distribution, i.e., identically distributed. This is the commonly used statistical assumption abbreviated as i.i.d. (independently, identically distributed) (Wei, 1990; Goodfellow et al., 2016). Each example used for training and testing has then the same probability distribution called the *data-generating distribution* denoted p_{data} (Goodfellow et al., 2016).

Goodfellow et al. (2016) states that the goodness of a learning algorithm is therefore often measured by its ability to:

1. Make the training error small
2. Make the gap between the training and test error small

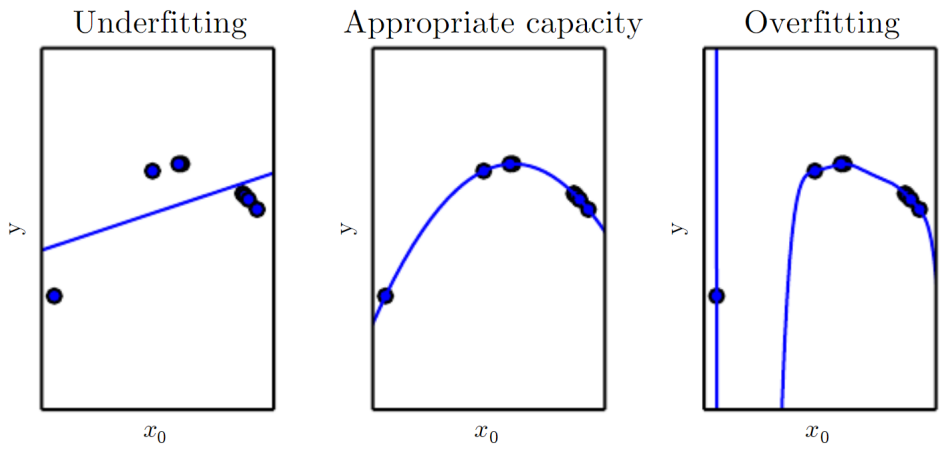


Figure 3.1: Idealized Examples of Machine Learning Fit (Goodfellow et al. (2016) Figure 5.5 Section 5.2)

These abilities are closely linked to *appropriate fitting*, which is one of the main challenges in machine learning (LeCun et al., 2015; Goodfellow et al., 2016). Figure 3.1, adapted from Goodfellow et al. (2016) Chapter 5, depicts the three idealized main cases of fitting which can occur.

Goodfellow et al. (2016) explains that the left figure shows the *overfitting* case, which is the result of excessive distance between the generalization and training error, i.e., too high variance. The right figure shows the *underfitting* case which is the result of insufficiently low training error, i.e., too high bias. Variance and bias will be further detailed in Section 3.2.2 below. The middle case depicts appropriate fitting.

3.2.1 Model Capacity

Goodfellow et al. (2016) defines the *capacity* of a model as its ability to effectively fit a range of functions and altering this determines whether the model is more likely to over- or underfit. The capacity of a model is not only determined by the choice of model family, but also by the corresponding design decisions related to the model family (Goodfellow et al., 2016).

Too low capacity generally lead to reduced ability to fit the training set, while too high capacity can cause the model to learn non-general and irrelevant patterns from the training set which does not extend to the test set (Goodfellow et al., 2016). The optimal capacity is therefore relative to the complexity of the task and the availability of training data (Goodfellow et al., 2016).

Figure 3.2, adapted from Goodfellow et al. (2016) Chapter 5, depicts the general development of the training and generalization error with increasing capacity. Goodfellow et al. (2016) explains that the generalization error normally decreases to a minima before it starts increasing again. This is due to that excessive capacity leads to overfitting and resulting poor generalization. The training error generally decreases with increasing capacity until it reaches a minima where it forms an asymptote. (Goodfellow et al., 2016) emphasizes that this requires that the model have some minima for the training error which may not always be the case.

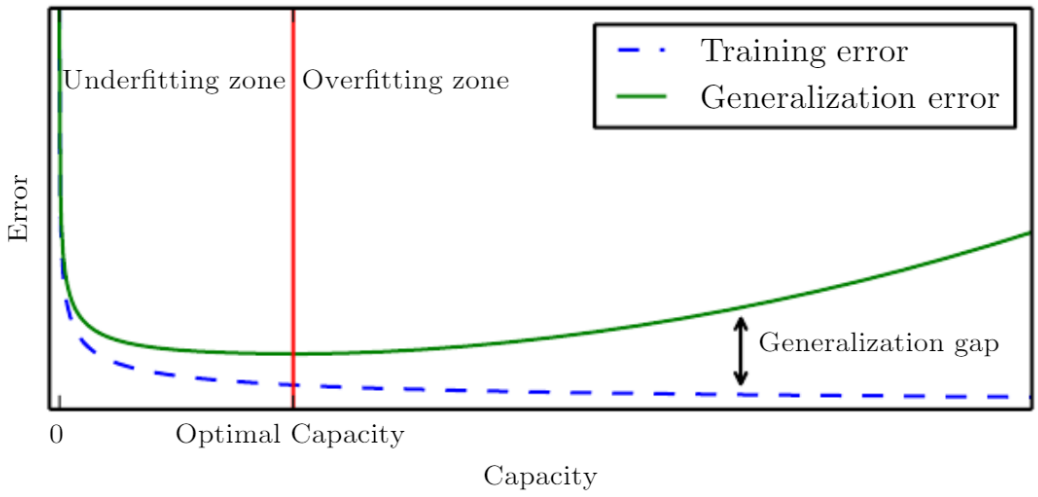


Figure 3.2: General Development of Training and Generalization Error as a Function of Capacity (Goodfellow et al. (2016) Figure 5.3 Section 5.2)

3.2.2 Bias and Variance in Learning

Goodfellow et al. (2016) states that estimators in machine learning are primarily point estimators. Lehmann and Casella (1998) defines a point estimator as a rule, which gives a single-valued result, for determining the estimate of a given quantity based on observed data. The quantity could be a single vector-valued result or any result given by a single function (Lehmann and Casella, 1998).

Goodfellow et al. (2016) gives a very general definition of a point estimator

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}) \tag{3.1}$$

where $\hat{\boldsymbol{\theta}}$ is the point estimate of the true value $\boldsymbol{\theta}$. The relationship between inputs and labels in a machine learning algorithm can be estimated by a point estimator (Goodfellow et al., 2016).

Goodfellow et al. (2016) further defines the bias of an estimator as

$$bias(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}_{data}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta} \tag{3.2}$$

An unbiased estimator is an estimator $\hat{\boldsymbol{\theta}}_m$ where $Bias(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$ (Goodfellow et al., 2016).

Goodfellow et al. (2016) states that the variance of an estimator determines how much it is expected to vary as a function of the data sample and is given as the variance of the training data as a random variable

$$Var(\hat{\boldsymbol{\theta}}) \tag{3.3}$$

Overfitting and underfitting is closely connected to bias and variance. Goodfellow et al. (2016) uses as an example the case of prediction algorithms, where the generalization error is often measured by the mean squared error (MSE), given by

$$MSE = \mathbb{E}[\hat{\boldsymbol{\theta}}_m - \boldsymbol{\theta}^2] = Bias(\hat{\boldsymbol{\theta}}_m)^2 + Var(\hat{\boldsymbol{\theta}}_m) \tag{3.4}$$

MSE measures a specific type of error between the estimator and the true value. Figure 3.3, adapted from Goodfellow et al. (2016) Chapter 5, depicts the close relationship between appropriate fitting, bias and variance. Figure 3.3 shows that increasing capacity generally decreases bias and increases variance. A good estimator has a low MSE and therefore must manage to avoid excessive variance or bias (Goodfellow et al., 2016).

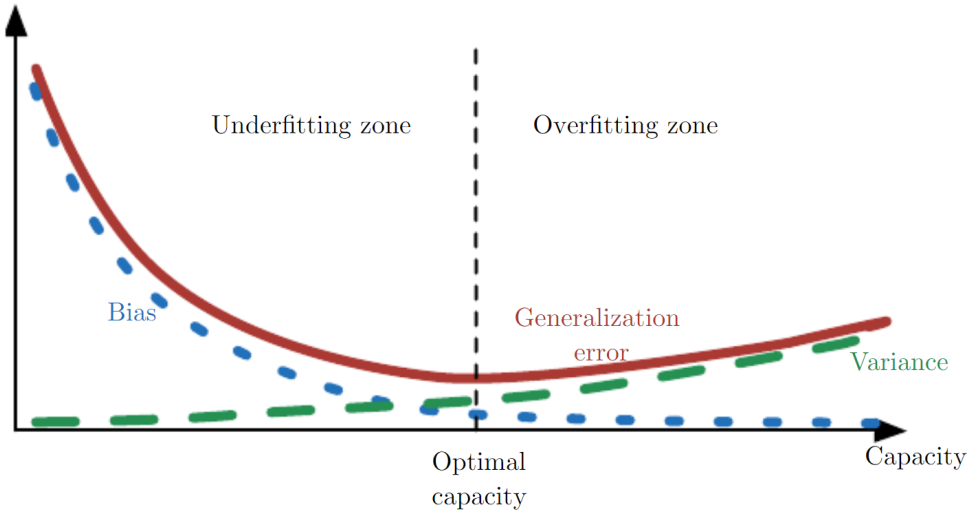


Figure 3.3: General Development of Bias and Variance as a Function of Capacity (Goodfellow et al. (2016) Figure 5.6 Section 5.4.4)

3.2.3 Hyperparameters and Datasets

Goodfellow et al. (2016) defines a *hyperparameter* as some setting of a machine learning algorithm that is determined *outside* the learning algorithm. These are in general not adjusted by the learning algorithm and control some aspect of the behavior of the algorithm (Goodfellow et al., 2016). Deep learning algorithms contain a wide range of different hyperparameters which will be detailed in the following sections.

Goodfellow et al. (2016) emphasizes that hyperparameters that determine the model capacity can not be learned on the training set as the algorithm would choose the maximum achievable capacity, which will result in overfitting.

Goodfellow et al. (2016) states that the dataset is therefore often further split into three disjoint sets:

- Training set
- Validation set
- Test set

The previously discussed test set must not be used for making any decision related to the algorithm, while the training set and validation set are used for just this purpose (Goodfellow et al., 2016). The combination of the training set and validation set is often referred to only as the training set, which understandably causes some confusion (Goodfellow et al., 2016). The training set is then composed of two subsets which comes from from the same data-generating process as the test set. Goodfellow et al. (2016) states that it is common for practitioners to split the training data by the 80-20 rule, 80 percent for training and 20 percent for validation. The

split between the full training set and test set is dependent on the practitioner and availability of data (LeCun et al., 2015; Brownlee, 2017).

Goodfellow et al. (2016) emphasizes that the generalization error on the validation set will be underestimated since it was used to find the correct setting of the hyperparameters. Therefore, after the hyperparameters are determined, the "true" generalization error can be estimated from the previously unseen test set (Goodfellow et al., 2016).

3.3 Feedforward Neural Networks

This section defines and details essential concepts for feedforward networks and is based on Chapter 6 in Goodfellow et al. (2016). Feedforward neural networks are presented since the concepts builds a foundation for understanding the more complex recurrent neural networks presented in Section 3.6.

Bengio et al. (2013) defines deep learning as a subset of the broader family of machine learning methods based on representation learning. Representation learning is a set of techniques that enables a system to automatically discover the representations needed when fed with training data (LeCun et al., 2015).

Goodfellow et al. (2016) separates deep learning methods from other representation models by that deep learning methods have multiple levels of representation, obtained by combining several simple, mostly non-linear modules. LeCun et al. (2015) explains that each module transform the representation at one level into a representation at a higher, slightly more abstract level. The combination of several such transformations enables deep learning models to learn very complex functions (LeCun et al., 2015). Deep methods have dramatically improved the state-of-the-art in speech recognition, natural language processing, image recognition and many other domains such as drug design (Ghasemi et al., 2017; Krizhevsky et al., 2012; LeCun et al., 2015).

According to Goodfellow et al. (2016), the most commonly used deep learning models are deep feedforward neural networks (FNN), also known as multilayer perceptrons (MLP). Goodfellow et al. (2016) states that a deep feedforward network defines the mapping $\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{\theta})$ which aims to approximate some function $f^*(\boldsymbol{x})$. The value of the parameters $\boldsymbol{\theta}$ are determined as those that result in the best approximation of $f^*(\boldsymbol{x})$. The determination of the parameters is referred to as training the network (LeCun et al., 2015).

Feedforward network derives its name from that information flows from $\boldsymbol{x}$ through the intermediate mapping $f(\boldsymbol{x};\boldsymbol{\theta})$ to the output $\boldsymbol{y}$ (Goodfellow et al., 2016). Hence, there are no connection which enables information to flow from $\boldsymbol{y}$ back into the model itself.

Goodfellow et al. (2016) states that a deep learning algorithm consists of the following components:

- A dataset
- A deep learning model
- A cost function
- An optimization method

These components will be detailed in the following sections.

3.3.1 Feedforward Neural Network Architecture

Goodfellow et al. (2016) present that deep neural networks are normally composed of several functions connected in chain structure. Using the notation and terminology of Goodfellow et al. (2016), let $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ be some functions connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. On this form, $f^{(1)}$ is called the first *layer* and $f^{(2)}$ is called the second layer, and so on. The final layer is called *output layer*. The inputs are in some cases denoted as its own layer, *input layer*, but it is important to note that the input layer is not associated with a function. The layers between the input and output layer are called *hidden layers*. The number of hidden layers determines the *depth* of the model.

Each layer consists of several *units*, also known as neurons. Each unit represents a vector-to-scalar function and the dimensionality of the hidden layers determines the *width* of the model (Goodfellow et al., 2016). These units compute their own activation value based on input from several other units. The activation value is given by the choice of *unit function*, also known as activation function. Goodfellow et al. (2016) divides unit functions into *hidden unit functions* and *output unit functions*, where the name refers to the position of the unit in the network. Unit functions are covered in detail in Section 3.3.3 and 3.3.4.

Figure 3.4 depicts a graph representation of a feedforward network with two hidden layers and a single output. The circles denote the units in each layer, while the arrows denotes the *weights*. Thus, the depth of the model is two and the width is five. The mathematical definition of weights is given below. Note that there are no units in the input layer.

Goodfellow et al. (2016) specifies that in each point of training, the output layer is directly controlled and must produce a value that is close to $\boldsymbol{y}$, while the hidden layers are not subject to such control. The learning algorithm must therefore decide on how to best use the hidden layers to implement an approximation of $f^*(\boldsymbol{x})$ (Goodfellow et al., 2016).

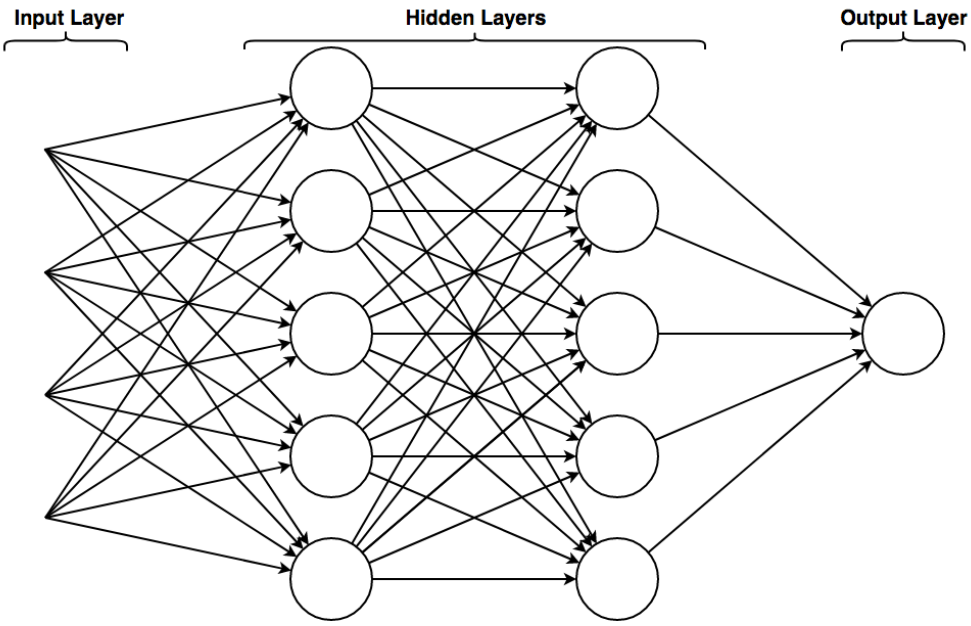


Figure 3.4: Graph Representation of Feedforward Network (Adapted from Nielsen (2015) Chapter 1)

Goodfellow et al. (2016) comments that deep learning can be understood as an extension of linear models. More formally, Goodfellow et al. (2016) explains that in order to represent nonlinear functions of $\boldsymbol{x}$ one can apply a linear model to a nonlinear transformation $\phi(\boldsymbol{x})$. Goodfellow et al. (2016) presents that deep learning aims to learn the mapping ϕ and the model thus becomes $y = f(\boldsymbol{x};\boldsymbol{\theta},\boldsymbol{w}) = \phi(\boldsymbol{x};\boldsymbol{\theta})^T\boldsymbol{w}$. The mapping ϕ is then learned from a class of functions using the parameters $\boldsymbol{\theta}$. The transform $\phi(\boldsymbol{x})$ maps to the desired output, given parameters $\boldsymbol{w}$. The hidden layer is here defined by the mapping ϕ and a good representation of $\phi(\boldsymbol{x};\boldsymbol{\theta})$ is then found by optimizing $\boldsymbol{\theta}$.

Continuing with the notation and approach in Goodfellow et al. (2016), let $\boldsymbol{\theta}$ be a linear function consisting of $\boldsymbol{w}$ and $\boldsymbol{b}$ and the choice of form for $f(\boldsymbol{x};\boldsymbol{\theta})$. Further, let $\boldsymbol{h}^{(1)}$ be a vector of the units in the *first* hidden layer, determined by a function $f^{(1)}(\boldsymbol{x};\boldsymbol{W}^{(1)},\boldsymbol{b}^{(1)})$. Goodfellow et al. (2016) remarks that most deep learning algorithms use the strategy of applying an affine transformation given by learned parameters $\boldsymbol{w}$ and $\boldsymbol{b}$, and then use a nonlinear activation function $g(\boldsymbol{z})$. Goodfellow et al. (2016) expresses the first hidden layer as

$$\boldsymbol{h}^{(1)} = g^{(1)}(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \tag{3.5}$$

where matrix $\boldsymbol{W}^{(i)}$ and vector $\boldsymbol{b}^{(i)}$ are the *weights* and the *biases* of a linear transformation.

Goodfellow et al. (2016) generalizes Equation 3.5 for hidden layer i as

$$\boldsymbol{h}^{(i)} = g^{(i)}(\boldsymbol{W}^{(i)T}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}) \tag{3.6}$$

where i is an integer larger than one.

To simplify the notation over several hidden layers, Goodfellow et al. (2016) lets matrix $\boldsymbol{W}$ denote the mapping from $\boldsymbol{x}$ to $\boldsymbol{h}$ and vector $\boldsymbol{w}$ denote the mapping from $\boldsymbol{h}$ to $\boldsymbol{y}$.

3.3.2 Cost Functions

Nielsen (2015) state that learning algorithms for deep networks are generally based on iterative, gradient-based optimizers that drive some cost function to a low value. The cost function computes some metric calculated from comparing the labels and values provided by the output layer (Nielsen, 2015). The exact application of the cost function will become clear in Section 3.4 when optimization procedures for deep learning are presented. This section serves as an introduction to cost functions for real-valued prediction tasks.

Most modern neural networks are trained using maximum likelihood since the parametric model in most cases defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ (Goodfellow et al., 2016). In that case, Goodfellow et al. (2016) defines the cost function as the negative log-likelihood

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} log\, p_{model}(\boldsymbol{y}|\boldsymbol{x}) \tag{3.7}$$

Thereby, the form of $log\, p_{model}$ determines the particular form of the cost function J. Goodfellow et al. (2016) notes that utilizing maximum likelihood is highly beneficial as the cost function $log\, p(\boldsymbol{y}|\boldsymbol{x})$ is automatically determined by specifying a model $p(\boldsymbol{y}|\boldsymbol{x})$.

In the context of prediction, Goodfellow et al. (2016) remarks that the cost function should be designed such that the minimum lie on the function that maps $\boldsymbol{x}$ to the expected value of $\boldsymbol{y}$ given $\boldsymbol{x}$. Nielsen (2015) and Goodfellow et al. (2016) states that two common cost function for prediction problems are mean square error (MSE) and mean absolute error (MAE), given in Equation 3.8 and 3.9, respectively.

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2 \tag{3.8}$$

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|_1 \tag{3.9}$$

Both functions are frequently used as cost functions in time series prediction applications, as seen in Brownlee (2017).

3.3.3 Output Unit Functions

The units in the output layer are associated with an output unit function which provide the values for the cost function (Nielsen, 2015). This means that the choice of output unit function significantly influences the choice of cost function and reverse (Nielsen, 2015). Goodfellow et al. (2016) states that the most frequently used output units are linear, sigmoid and rectified linear outputs. Nielsen (2015) remarks that any output unit function can also be used for the hidden units.

Linear Units

Goodfellow et al. (2016) defines a linear output unit as the affine transformation

$$\hat{\boldsymbol{y}} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b} \tag{3.10}$$

Goodfellow et al. (2016) states that a common application of linear output layers is to generate the mean of a conditional Gaussian distribution, $p(\boldsymbol{y}|\boldsymbol{x}) = N(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$. A benefit of linear units is that the algorithm is likely to learn the covariance of a Gaussian or to make the covariance a function of the input, due to the maximum likelihood approach (Goodfellow et al., 2016).

Sigmoid Units

Sigmoid units are often applied when the goal is predicting the value of a binary variable y, i.e., Bernoulli output distributions (Goodfellow et al., 2016). Goodfellow et al. (2016) defines a sigmoid unit as

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{3.11}$$

where the sigmoid unit first computes $z = \boldsymbol{W}^T \boldsymbol{h} + b$, similar to a linear output unit. The output of a sigmoid unit thus becomes a probability.

Softmax Units

Softmax units are a generalization of sigmoid units and are commonly used when predicting the probability distribution over a discrete variable with n possible values, i.e, Multinoulli output distributions (Goodfellow et al., 2016). Goodfellow et al. (2016) defines softmax units as a unit which first predicts $\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{h} + \boldsymbol{b}$, where $z_i = log\tilde{P}(y = i|\boldsymbol{x})$, and then applies the softmax function

$$softmax(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)} \tag{3.12}$$

The output is then a probability vector $\hat{\boldsymbol{y}}$, where $\hat{y}_i = P(y = i|\boldsymbol{x})$ (Goodfellow et al., 2016).

3.3.4 Hidden Unit Functions

Most hidden units takes an input $\boldsymbol{x}$ or $\boldsymbol{h}$, dependent on the location of the layer, and computes the affine transformation and then apply some element-wise nonlinear activation function $g(z)$ (Goodfellow et al., 2016).

Rectified Linear Units (ReLU)

Nair and Hinton (2010) defines the rectified linear activation function as

$$g(z) = max\{0, z\} \tag{3.13}$$

ReLU is the preferred default choice of hidden units for several model families and applications (Goodfellow et al., 2016).

Hyperbolic Tangent Unit Function

Goodfellow et al. (2016) states that the hyperbolic tangent activation function is simply given by

$$g(z) = tanh(z) \tag{3.14}$$

3.4 Optimization of Deep Learning Algorithms

This section is based on the methods detailed in Goodfellow et al. (2016) Chapter 8. Goodfellow et al. (2016) emphasizes that optimization is a central part of deep learning since problems can require days to months of training, on hundreds of machines. This section focuses on the frequently used stochastic gradient descent optimization technique and the Adam extension.

3.4.1 Batch and Minibatch algorithms

Goodfellow et al. (2016) specifies that training optimization differs significantly from traditional optimization. Traditional optimization has a direct approach, it aims to minimize some cost function $J(\theta)$, while in training optimization it is common to have some performance criteria P that must be indirectly optimized by minimizing a different (cost) function $J(\theta)$.

Goodfellow et al. (2016) further states that another major difference from pure optimization is that the parameters in a machine learning algorithm are updated on a subset of the terms of the full cost function to compute an expected value of the cost function. Maximum likelihood estimation problems are, as previously explained, prevalent in deep learning. Goodfellow et al. (2016) states that these problems can be decomposed into

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} log\, p_{model}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{3.15}$$

Goodfellow et al. (2016) further shows that maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} log\, p_{model}(\boldsymbol{x}, y; \boldsymbol{\theta}) \tag{3.16}$$

The most commonly used property of the cost function $\boldsymbol{J}$ is the gradient which Goodfellow et al. (2016) states can also found as an expectation of the training set

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim\hat{p}_{data}} \nabla_{\boldsymbol{\theta}} log\, p_{model}(\boldsymbol{x}, y; \boldsymbol{\theta}) \tag{3.17}$$

Using every example in the dataset to compute exact expectations is computationally expensive and in practice the expectations are computed by the average of a randomly sampled subset of the dataset (Goodfellow et al., 2016).

Goodfellow et al. (2016) presents a classification of optimization algorithms by the number of examples they process simultaneously:

- Batch or deterministic methods: Process all training examples
- Stochastic or online methods: Process one training example
- Minibatch or minibatch stochastic methods: Process more than one and less then all training examples

Minibatch stochastic methods are most prevalent and are commonly referred to only as stochastic methods (Goodfellow et al., 2016).

Goodfellow et al. (2016) states that in practice the batch size is between 32- 512 examples and Keskar et al. (2016) found that larger batch sizes lead to a significant degradation in the ability of models to generalize. Further, Wilson and Martinez (2013) found that small batches can offer a regularizing effect and the generalization error is often lowest for a batch size of one. Small batch sizes increases the number of steps and requires a small *learning rate*, thus the total runtime can become very high (Wilson and Martinez, 2013). Learning rate will be formally defined in Section 3.4.3.

3.4.2 Epochs and Iterations

Before presenting the optimization algorithms, a separation must be made between the terms *iteration* and *epoch* in machine learning. An epoch is a complete pass through the training set, while an iteration is simply one update of the learning model's parameters (Patterson and Gibson, 2017). Hence, when using stochastic and minibatch methods, several iterations will take place during one epoch. Iteration and epoch are indistinguishable for batch methods, i.e., updating the parameters after processing all training data at once.

3.4.3 Stochastic Gradient Descent Algorithm

Stochastic gradient descent (SDG) and its extensions are the optimization methods of choice for machine learning and especially for deep learning (Goodfellow et al., 2016). Robbins and Monro (1951) first presented the method and the full algorithm, as given in Goodfellow et al. (2016) Chapter 8, can be found in Algorithm 1 below.

The algorithm, as explained by Goodfellow et al. (2016), first samples a minibatch of m independently and identically distributed (i.i.d) examples from the data-generating distribution. The minibatch is then used to find an unbiased estimate of the gradient $\boldsymbol{g}$ by taking the average of the gradient of the minibatch. The algorithm then follows the gradient downhill dependent on the value of the learning rate ϵ. To simplify the notation, Goodfellow et al. (2016) lets $\boldsymbol{L}$ denote the per-example loss $\boldsymbol{L}(\boldsymbol{x}, y, \boldsymbol{\theta}) = -log\, p(y|\boldsymbol{x}; \boldsymbol{\theta})$.

Algorithm 1 Stochastic Gradient Descent (SDG) (Goodfellow et al. (2016) Algorithm 8.1 Section 8.3.1)

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with targets $\boldsymbol{y}^{(i)}$
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \boldsymbol{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while

The learning rate is another hyperparameter which must be chosen before training and is often either fixed or set to gradually decay over time (Goodfellow et al., 2016). The value of the learning rate has a significant influence on the quality of the model and is most often chosen by trail and error (Nielsen, 2015). One approach is to monitor the objective function as function of time, but as Goodfellow et al. (2016) so eloquently puts it:

> *"This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism."*

Goodfellow et al. (2016) remarks that stochastic gradient descent often finds a very low value of the cost function, but it is not guaranteed to find a local minima in a reasonable amount of time. Despite this, the low value of the cost function is found quickly enough for the algorithm to be very useful (Goodfellow et al., 2016). SGD has the important property that the computation time for each update does not increase with the number of training examples (Nielsen, 2015).

3.4.4 Adam Algorithm

Adaptive learning rate algorithms were introduced to handle the complexity of determining the learning rate, as it is one of the most complicated hyperparameters to determine due to its significant effect on model performance (Goodfellow et al., 2016). Most of these algorithms can be seen as extensions of stochastic gradient descent. Kingma and Ba (2014) presented such an adaptive algorithm called Adam (adaptive moments). Adam combines parts of a previous adaptive learning algorithm called RMSProp (Hinton, 2012) with the concept of momentum (Goodfellow et al., 2016). Momentum drives the algorithm to move in the direction of the accumulation of an exponentially decaying moving average of past gradients (Goodfellow et al., 2016). Momentum improves learning when dealing with high curvature, noisy gradients or small but consistent gradients (Hinton, 2012).

It is sometimes necessary to adjust the learning rate of Adam from the suggested default, but it is generally regarded as being quite robust to the choice of hyperparameters (Goodfellow et al., 2016). The full algorithm, as given in Goodfellow et al. (2016) Chapter 8, can be found in Algorithm 2 below.

Algorithm 2 Adam Algorithm (Goodfellow et al. (2016) Algorithm 8.7 Section 8.5.4)

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_1 in [0,1). (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}, \boldsymbol{r} = \mathbf{0}$
  Initialize time step t = 0
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with targets $\boldsymbol{y}^{(i)}$
    Compute gradient: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i \boldsymbol{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \Delta\boldsymbol{\theta}$
    $k \leftarrow k + 1$
  end while

3.4.5 Back-propagation Algorithm for Computing the Gradient

Adam and most other learning algorithms require the gradient of the cost function (Hinton, 2012; Kingma and Ba, 2014; Goodfellow et al., 2016). The gradient is in nearly all cases computed by the back-propagation (backprop) algorithm (Rumelhart et al., 1986).

(Goodfellow et al., 2016) defines forward propagation as the process in which information flows forward through a network, from input $\boldsymbol{x}$ to output $\hat{\boldsymbol{y}}$. The forward propagation generally continues until it results in a scalar cost $\boldsymbol{J}(\boldsymbol{\theta})$ (Goodfellow et al., 2016). The idea of back-propagation is then to let the information flow back through the network and use this to compute the gradient (Rumelhart et al., 1986).

The full algorithm will not be presented here as is not necessary within the scope of the thesis. It is sufficient to understand that the backprop algorithm can efficiently compute the gradient. The original algorithm can be found in Rumelhart et al. (1986).

3.4.6 Optimization Algorithm Selection

Goodfellow et al. (2016) states that there is currently no consensus for selecting the "right" optimization algorithm. Schaul et al. (2013) tested a wide range of optimization algorithms on different learning tasks and found that the adaptive learning rate algorithms performed fairly robust. No clear winner was chosen and most popular are SGD, RMSProp, AdaDelta (Duchi

et al., 2011) and Adam. Goodfellow et al. (2016) remarks that the choice of algorithm depends mostly on the user familiarity with tuning the hyperparameters of the algorithm.

3.5 Regularization

This section is based on Chapter 7 in Goodfellow et al. (2016), where Goodfellow et al. (2016) starts with defining *regularization* as:

> *"any modification we make to a learning algorithm that is intended to reduce its generalization error, but not its training error"*

The goal of an effective regularizer is then to reduce the variance significantly, while not at the same time increasing the bias too much (Goodfellow et al., 2016). This should increase the model's ability to generalize for new data without underfitting (Goodfellow et al., 2016).

3.5.1 Parameter Norm Penalties (L^1 and L^2)

LeCun et al. (2015) states that several regularization methods utilize the strategy of limiting the capacity of the deep learning model by adding a parameter norm penalty term $\Omega(\boldsymbol{\theta})$ to the cost function $\boldsymbol{J}$. Goodfellow et al. (2016) defines the regularized cost function $\tilde{\boldsymbol{J}}$ as

$$\tilde{\boldsymbol{J}}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = \boldsymbol{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}) \tag{3.18}$$

where the norm penalty term Ω is weighted by the hyperparameter $\alpha \in [0, \infty)$. Exactly what Ω penalizes is dependent on the algorithm design and model family (Goodfellow et al., 2016). This section focus on regularizers which only penalizes the weights of the affine transformation from layer to layer. The bias is thereby left unregulated, but biases generally require less data to be fit accurately (Goodfellow et al., 2016).

L^2 regularization, also known as weight decay or ridge regression, is the most common form of parameter norm penalty regularizer (Goodfellow et al., 2016). In the notation of Goodfellow et al. (2016), let $\boldsymbol{w}$ denote all the weights in $\boldsymbol{\theta}$ affected by a norm penalty. Goodfellow et al. (2016) then defines the L^2 norm penalty as

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{3.19}$$

A learning algorithm optimizing a cost function with L^2 regularization will shrink the weights on features which have a low covariance with the output target compared to the added variance (Goodfellow et al., 2016).

Goodfellow et al. (2016) defines another common norm penalty L^1, also known as lasso, as

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_1 = \sum_i |w_i| \tag{3.20}$$

L^1 regularization results in a solution that is more sparse. The regularization drives some of the parameters to have an optimal value of zero (Goodfellow et al., 2016).

Zou and Hastie (2005) combined the L^1 and L^2 norm penalty regularizers to form a norm penalty function called elastic net

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_1 + \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{3.21}$$

Zou and Hastie (2005) states that elastic net overcomes some of the drawbacks associated with L^1 regularization. If the number of examples is a small integer n and the dimension of the data is much larger than n, then L^1 becomes saturated when selecting at most n variables Zou and Hastie (2005). L^1 also tends to only select one variable in a group of highly correlated variables, ignoring the rest (Zou and Hastie, 2005).

Zou and Hastie (2005) showed that in an elastic net, L^1 generates a sparse model while L^2 removes the limit on selected variables, allows for grouping and stabilizes the regularization path.

3.5.2 Dropout

Srivastava et al. (2014) introduced the concept of dropout regularization, which is a very powerful method for regularizing several model types and has the great benefit of being computationally inexpensive. Srivastava et al. (2014) showed that dropout is more effective than other regularization methods and Goodfellow et al. (2016) notes that dropout can effectively be combined with other regularizers for further improvements.

Goodfellow et al. (2016) explains that the essence of dropout is to train the set of all subnetworks that can be formed by temporarily disabling hidden or input units from the total network under consideration. This can be achieved by multiplying the output of a unit by zero since most modern neural networks are, as detailed above, formed by a series of affine transformations and non-linear functions (Srivastava et al., 2014).

More formally, Goodfellow et al. (2016) defines dropout by the following procedure. Let $\boldsymbol{\mu}$ be a mask vector that specifies which units to include and $\boldsymbol{J}(\boldsymbol{\theta}, \boldsymbol{\mu})$ be the cost of a model defined by parameters $\boldsymbol{\theta}$ and mask $\boldsymbol{\mu}$. Training with dropout relies then on minimizing $\mathbb{E}_{\boldsymbol{\mu}}\boldsymbol{J}(\boldsymbol{\theta}, \boldsymbol{\mu})$. The binary mask is sampled for each example into a mini-batch and then applied to all the hidden or input units in the network.

In practice, values of $\boldsymbol{\mu}$ are sampled to obtain an unbiased estimate of the gradient of $\boldsymbol{J}$ since the expectation contains exponentially many terms (Srivastava et al., 2014). The sampling is independent for each unit and the probability of a unit being included is a hyperparameter determined before training (Goodfellow et al., 2016). An inclusion probability of 50 percent and 80 percent is often used for hidden and input units, respectively (Goodfellow et al., 2016).

3.5.3 Early Stopping

Early stopping is another regularization strategy that is highly popular due to its simplicity and effectiveness (Goodfellow et al., 2016). Recall the general development of the validation error as shown in Figure 3.2 above. The validation error decreases at first and after reaching a minima it steadily starts to rise again. Goodfellow et al. (2016) remarks that this happens frequently in deep learning.

Goodfellow et al. (2016) states that the strategy of early stopping is simply to save the model parameters each time there is an improvement in the validation error and then returning the best parameters after training is completed. In some applications, the training processes is terminated if there is no improvement for a set number of iterations or epochs (Chollet, 2015). Early stopping can be combined effectively with other regularization strategies (Goodfellow et al., 2016).

3.6 Recurrent Neural Networks

Goodfellow et al. (2016) defines recurrent neural networks (RNN) as a family of models specialized for processing a sequence of values $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(\tau)}$. Recurrent networks are scalable with longer sequences than practical for other non-specialized networks and have the ability to process sequences of different length (Goodfellow et al., 2016).

Goodfellow et al. (2016) states that the main benefit of recurrent networks is that they utilize parameters sharing across the model, where each member of the output is given by applying the same update rule to the previous members of the output. Goodfellow et al. (2016) further states that long short-term memory (LSTM) networks are the most effective recurrent models used in practical applications. LSTM networks were first presented in Hochreiter and Schmidhuber (1997).

The section is based in Chapter 10 in Goodfellow et al. (2016) and starts with the basic concepts of unfolding computational graphs before presenting recurrent neural networks with emphasis on LSTM networks.

3.6.1 Computational Graphs and Unfolding

Goodfellow et al. (2016) defines computational graphs as a directed graph where each node correspond to a variable, where the variable can be of many types, such as but not limited to vector, matrix, tensor or scalar. In addition to the nodes, there is a set of allowed operations which are simple functions of one or more variable and more complex operations are built by combining the allowed set of simple operations (Goodfellow et al., 2016). Computational graphs are commonly used to structure a set of computations (Goodfellow et al., 2016).

Goodfellow et al. (2016) gives the classical form of a recurrent dynamic system

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta}) \tag{3.22}$$

where $\boldsymbol{s}^{(t)}$ is the state of the system at time t. The recurrence comes from that the definition $\boldsymbol{s}$ refers back to the same definition at time $t-1$ (Goodfellow et al., 2016). This recurrent computation can be unfolded into a repetitive computational graph and for a given number of time step τ, the graph can be unfolded $\tau-1$ times by repeating the definition in Equation 3.22 (Goodfellow et al., 2016). Goodfellow et al. (2016) shows that for $\tau = 3$ the unfolding becomes

$$\boldsymbol{s}^{(3)} = f(\boldsymbol{s}^{(2)};\boldsymbol{\theta}) = f(f(\boldsymbol{s}^{(1)};\boldsymbol{\theta});\boldsymbol{\theta}) \tag{3.23}$$

Goodfellow et al. (2016) states that since Equation 3.23 no longer involve any recurrence it can therefore be represented as a traditional directed acyclic graph. The same approach is commonly used for values of hidden units, where the state is here the hidden unit (Goodfellow et al., 2016).

Let $\boldsymbol{x}^{(t)}$ be an input signal and $\boldsymbol{h}^{(t)}$ be the state, then Goodfellow et al. (2016) states that 3.22 becomes

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)};\boldsymbol{\theta}) \tag{3.24}$$

Goodfellow et al. (2016) explains that in prediction algorithms, the recurrent model normally learns a lossy summary of the relevant aspects of the previous input sequence. It becomes lossy since the fixed length vector $\boldsymbol{h}^{(t)}$ is mapped from the non-fixed length input sequence. The summary often contain some particular characteristic of the past sequence with a high precision compared to other characteristics.

Goodfellow et al. (2016) defines the function $g^{(t)}$ to represent the unfolded recurrence after t time steps

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)};\boldsymbol{\theta}) \tag{3.25}$$

The past sequence $(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)})$ is used as an input to function $g^{(t)}$ which produces the current state and function $g^{(t)}$ can further be factorized by repeatedly applying the function f (Goodfellow et al., 2016).

Goodfellow et al. (2016) explains that the unfolding, which gives the parameter sharing, results in two major advantages:

1. The input size of the model is constant as it is given in terms of transition from one state to another.

2. The same parameters are used with the same function f at all time steps.

These components enable the algorithm to learn a single shared model f that allows for generalization to sequence lengths that did not appear in the training set (Goodfellow et al., 2016). Far more training examples would be required if the model did not share parameters, as the model would need to learn a new model $g^{(t)}$ for all possible time steps (Goodfellow et al., 2016).

Figure 3.5, adapted from Goodfellow et al. (2016) Chapter 10, depict two common ways to represent recurrent models, unfolded computational graph and circuit diagram. The left figure shows

a circuit diagram where the black square denotes a single time step delay. The right figure shows the unfolded graph representation of the same network. Here, each node is connected to a specific time instance.

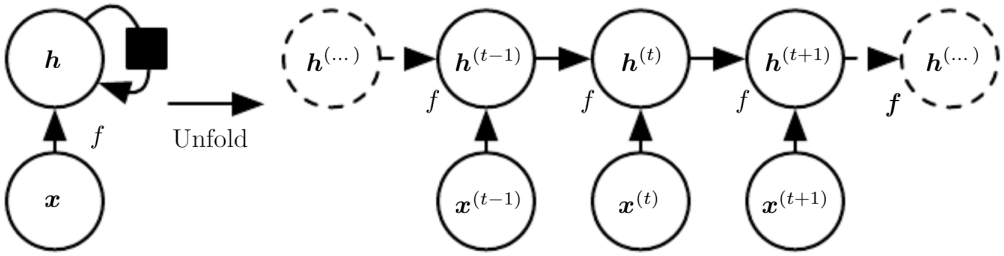


Figure 3.5: Unfolded Computational Graph and Circuit Diagram Representation (Goodfellow et al. (2016) Figure 10.2 Section 10.1)

3.6.2 General Recurrent Neural Network

Recurrent networks can be configured in several different ways. Figure 3.6, from Goodfellow et al. (2016) Chapter 10, depicts the unfolded computational graph and circuit diagram of a recurrent network which have recurrent connections between hidden units and for each time step outputs a value.

Goodfellow et al. (2016) explains that the network in Figure 3.6 takes an input sequence $\boldsymbol{x}$ and maps it to a corresponding sequence of outputs $\boldsymbol{o}$. The distance between the output sequence $\boldsymbol{o}$ and the training label $\boldsymbol{y}$ is measured by the loss $\boldsymbol{L}$. Depending on the choice of output unit function and cost function, the loss $\boldsymbol{y}$ computes some loss metric from the output $\boldsymbol{o}$ and label $\boldsymbol{y}$. The weight matrix $\boldsymbol{U}$ parametrizes the input-to-hidden connections, while matrix $\boldsymbol{V}$ parametrizes the hidden-to-output connections.

This is a general depiction of a simple recurrent network and thus the loss function and hidden units are not specified. Recurrent neural networks utilize several different configurations and functions dependent on the task (Goodfellow et al., 2016).

3.6.3 The Gradient in Recurrent Neural Networks

Computing the Gradient in Recurrent Network

Determining the gradient in a recurrent network does not differ from that of a feedforward network (Goodfellow et al., 2016). The generalized back-prop algorithm described in Section 3.4.5 can be applied to the unfolded computational graph (Goodfellow et al., 2016). Any general-purpose gradient-based optimization method may use the gradient to train the recurrent network, such as the previously discussed SDG and Adam (Goodfellow et al., 2016; Kingma and Ba, 2014; Robbins and Monro, 1951).

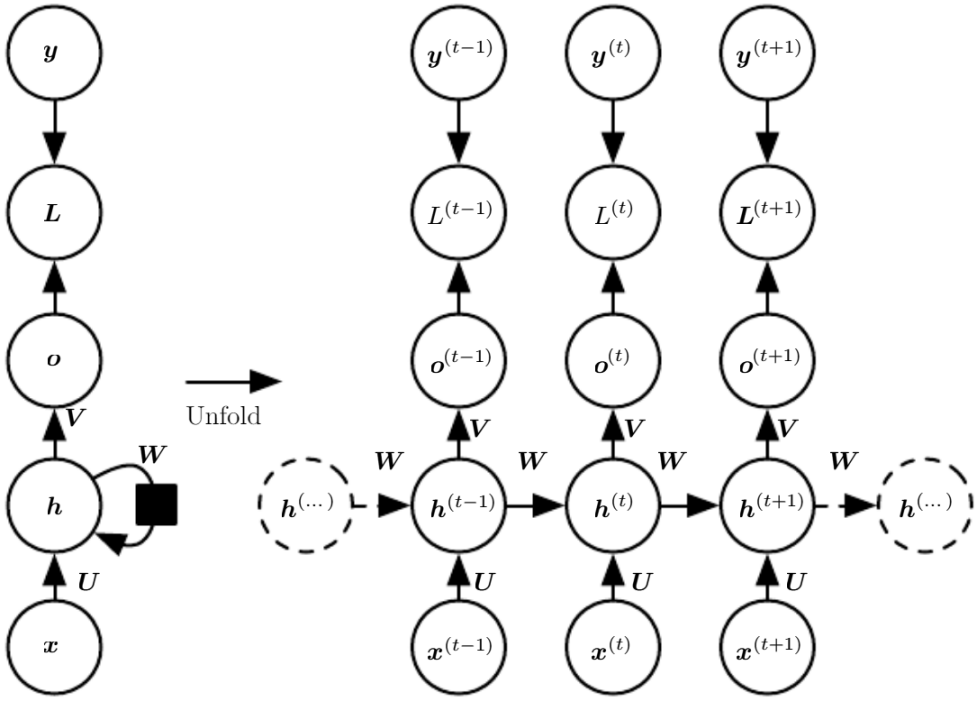


Figure 3.6: Circuit Diagram and Unfolded Computational Graph of General Recurrent Network (Goodfellow et al. (2016) Figure 10.3 Section 10.2)

Long-term Dependencies in Recurrent Networks

Goodfellow et al. (2016) states that the main challenge with learning long-term dependencies in recurrent networks comes from that gradients flowing through many stages tend to either explode or vanish. Gradients vanish much more frequently than they explode (Goodfellow et al., 2016).

Goodfellow et al. (2016) explains that the problem arises from the fact that the long-term interaction gradient has a magnitude that is exponentially smaller than the short-term gradient. This leads to a tremendous training time for learning long-term dependencies as these dependencies tend to be hidden by even minor fluctuations in short-term dependencies (Goodfellow et al., 2016).

3.6.4 Long Short-term Memory (LSTM) Network

Long short-term memory models introduced the concept of self-loops (Hochreiter and Schmidhuber, 1997). The self-loops helps avoid the vanishing or exploding gradient problem by generating paths where the gradient can flow for a long duration (Goodfellow et al., 2016). This allows for a model that can handle fine details in the immediate past, while at the same time efficiently transfer relevant information from the far past (Goodfellow et al., 2016).

Goodfellow et al. (2016) explains that LSTM models deviate from feedforward networks in that there is no longer a unit, which computes the affine transformation of inputs, and then apply a

nonlinear function. Instead, LSTM networks have special type of cell (layer) called LSTM cells. These LSTM cells have a self-loop, internal recurrence, combined with an additional outer recurrence.

Each cell is connected to three types of gating units and the flow of information in the network is controlled by these gating units (Goodfellow et al., 2016). Using the notation and method as described in Goodfellow et al. (2016) let $s_i^{(t)}$ denote the state unit in cell i at time step t. A forget gate $f_i^{(t)}$ controls the self-loop weight, through a sigmoid unit, by setting the corresponding weight to a value between 0 and 1. Goodfellow et al. (2016) defines the forget gate function as

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}) \tag{3.26}$$

The current input and hidden layer vector is denoted $\boldsymbol{x}^{(t)}$ and $\boldsymbol{h}^{(t)}$, respectively. The hidden layer vector contains the entirety of the LSTM cell's outputs. The total weights of the forget gate are split into recurrent weights $\boldsymbol{U}^f$ and input weights $\boldsymbol{W}^f$. The bias of the forget gate is denoted $\boldsymbol{b}^f$.

Goodfellow et al. (2016) gives the function for second type of gate unit, external input gate $g_i^{(t)}$, as

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}) \tag{3.27}$$

The superscript g here denotes that the weights and biases belong to the internal input gates.

Goodfellow et al. (2016) states that the update of the internal state of the of the LSTM cell is then given by

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}) \tag{3.28}$$

Note that the conditional self-loop, given in Equation 3.26 above, is included in the expression and multiplied with the previous state unit, thus acting as a self-controlled weight. The same applies to the external input gate, but it controls the sigmoid value into the LSTM cell.

The final type of gate, output gate $q_i^{(t)}$, is used to control the output of the LSTM cell through another sigmoid function

$$q_i^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}) \tag{3.29}$$

where the superscript o denotes that the weights and biases belong to output gate unit (Goodfellow et al., 2016).

The function for the output gate of the LSTM cell is given by Goodfellow et al. (2016) as

$$h_i^{(t)} = tanh(s_i^{(t)}) q_i^{(t)} \tag{3.30}$$

Figure 3.7, adapted from Goodfellow et al. (2016) Chapter 10, shows a block diagram of a LSTM cell.

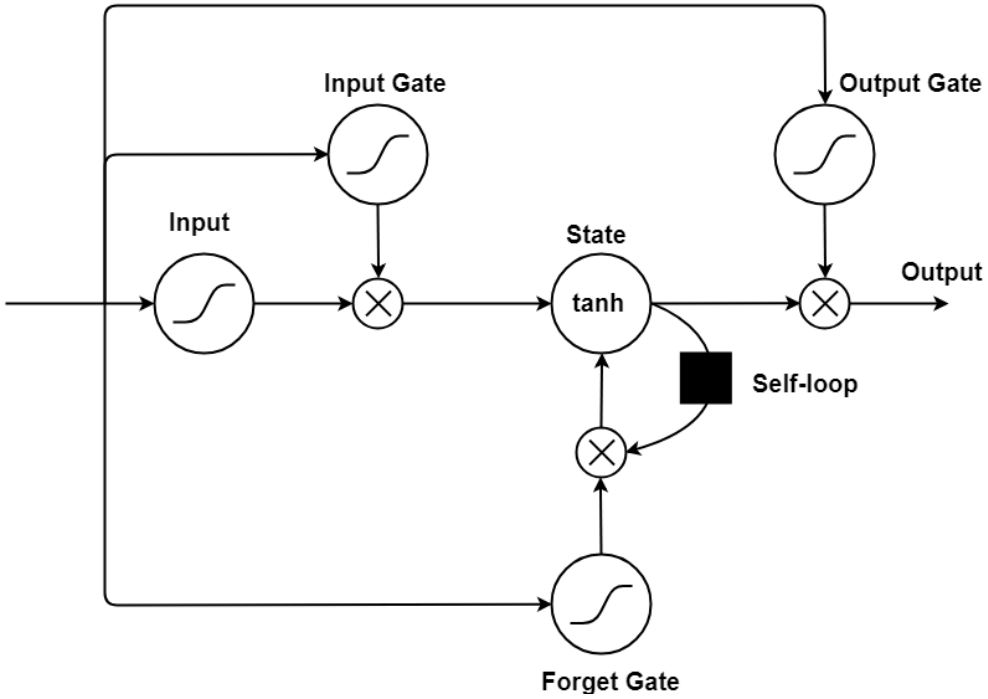


Figure 3.7: Block Diagram of LSTM Network Cell (Adapted from Goodfellow et al. (2016) Figure 10.16 Section 10.10.1)

Goodfellow et al. (2016) explains that the input gate in Figure 3.7 uses a sigmoid unit to compute a value from the inputs which can be accumulated to the state depending on the sigmoid value of the input gate. The sigmoid value of the forget gate controls the linear self-loop of the state unit, where the black square denotes a single time step. The output from the state is controlled by the sigmoid value of the output gate. The self-loop results in an unfolding similar, but more complex due to the added connections, to the unfolding of the general recurrent neural network as shown in Figure 3.6 (Goodfellow et al., 2016).

Stacking LSTM Layers

Pascanu et al. (2013) introduced that LSTM networks can be used with a deeper architecture by connecting together, often referred to as *stacking* (Brownlee, 2017), multiple LSTM cells. The output sequence of a LSTM cell is then returned to the next cell until the output layer. Graves et al. (2013) showed that increased depth improve model performance on several tasks. As with feedforward networks, the additional layers provide increased levels of abstraction, thus gives the possibility to further represent the problem at different time scales (Pascanu et al., 2013).

Chapter 4

Traditional Methods for Metocean Simulation

This chapter introduces the theory of two traditional methods used for metocean simulation: Vector autoregressive moving-average (VARMA) models and discrete-time finite-state Markov chain models. These models were chosen as they are frequently applied as stochastic generators in metocean simulation (Monbet et al., 2007). The models are therefore suitable comparison models for a deep learning stochastic metocean generator.

4.1 Vector Autoregressive Moving-average Models

The models in this section are from Wei (1990), but the notation follows Percival (1993). The change in notation was due to better readability and for maintaining notational consistency in the thesis.

4.1.1 Autoregressive (AR) Model

Wei (1990) defines the auto-regressive AR(p) model of order p as

$$X_t = c + \sum_{i=1}^{p} \phi_i X_{t-1} + \epsilon_t \tag{4.1}$$

where ϕ_i are the AR parameters, c is a constant and ϵ_t is white noise. Wei (1990) states that AR models are useful for modeling processes where the present value depends linearly on preceding values in addition to a random shock. The model is stationary if the roots of the polynomial $z^p - \sum_{i=1}^{p} \phi_i z^{p-i}$ lie outside the unit circle (Wei, 1990).

4.1.2 Moving-average (MA) Model

Wei (1990) defines the moving-average MA(q) model of order q as

$$X_t = \epsilon_t + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{4.2}$$

where θ_i are the MA parameters and $\epsilon_t, \epsilon_{t-1}, \ldots$ are white noise (Wei, 1990). The moving-average model is always stationary (Wei, 1990).

4.1.3 Vector Autoregressive Moving-average (VARMA) model

Wei (1990) defines the univariate ARMA(p, q) model with orders p, q as

$$X_t = c + \epsilon_t + \sum_{i=1}^{p} \phi_i X_{t-1} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} \tag{4.3}$$

The ARMA model is thus simply a composite of the AR and MA model. It should be noted that it is common in the literature to refer to the order of p and q as number of p and q lags (Wei, 1990).

Wei (1990) states that all of the models presented in this section can be extended to the multivariate case. The multivariate ARMA is often called vector ARMA (VARMA) and defined by Wei (1990) as

$$\boldsymbol{X}_t = \boldsymbol{c} + \boldsymbol{\epsilon}_t + \sum_{i=1}^{p} \boldsymbol{\phi}_i \boldsymbol{X}_{t-1} + \sum_{i=1}^{q} \boldsymbol{\theta}_i \boldsymbol{\epsilon}_{t-i} \tag{4.4}$$

where the AR and MA parameters $\boldsymbol{\phi}_i$ and $\boldsymbol{\theta}_i$ are $N \times N$ matrices and N is the number of variables. The remaining terms are $N \times 1$ vectors.

Wei (1990) and Percival (1993) both shorten and simplify the notation by introducing the lag operator L. Wei (1990) defines the lag operator as an operator which shifts an element given the integer it is raised to

$$L^k X_t = X_{t+k} \tag{4.5}$$

Using the lag operator, Wei (1990) states that the VARMA model can be written as

$$\boldsymbol{\Phi}(L)\boldsymbol{X}_t = \boldsymbol{c} + \boldsymbol{\Theta}(L)\boldsymbol{\epsilon}_t \tag{4.6}$$

where the AR component is given by

$$\boldsymbol{\Phi}(L) = 1 - \sum_{i=1}^{p} \boldsymbol{\phi}_i L^i \tag{4.7}$$

and the MA component is given by

$$\boldsymbol{\Theta}(L) = 1 + \sum_{i=1}^{q} \boldsymbol{\theta}_i L^i \tag{4.8}$$

Only the AR part influences stationary and a multivariate VARMA(p, q) is stationary if the roots of the polynomial $\boldsymbol{\Phi}(L)$ lie outside the unit circle (Wei, 1990).

Wei (1990) states that the parameters of a VARMA model are generally found by optimization methods based on maximizing the log-likelihood while ensuring stationarity. The exact procedure is not presented here, but the maximum likelihood estimation method and other estimation methods can found in Wei (1990) Chapter 7. The parameter estimation procedure estimates

the most likely parameters from the data which is to be modeled and is commonly known as fitting the model (Wei, 1990).

4.1.4 White Noise

Wei (1990) states that the white noise is generally assumed to be independent identically distributed (i.i.d.) random variables. Wei (1990) further remarks that the white noise is generally sampled from a zero mean normal distribution $\epsilon \sim \mathcal{N}(0, \sigma^2)$. Weakening these assumptions, in particular the i.i.d. assumption, would lead to fundamental difference in the model (Wei, 1990).

4.2 Discrete-Time, Finite-State Markov Chains

Kirkwood (2015) defines a discrete Markov chain or Markov process as a discrete stochastic process which satisfies the Markov property. The Markov property is also known as the memory-less property and Kirkwood (2015) states it as:

> "for a Markov process, conditioning a future event on a given set of previous outcomes is equivalent to conditioning only on the most recent of the outcomes in the set."

Let S denote the discrete and finite state space of a Markov chain and X is a random time indexed variable. Kirkwood (2015) then formally defines the Markov property as

$$P(X_{n+m} = s | X_{n-1} = i_{n-1}, \ldots, X_1 = i_1, X_0 = i_0) = P(X_{n+m} = s | X_{n-1} = i_{n-1}) \tag{4.9}$$

where $s, i_0, i_1, \ldots, i_{n-1} \in S$ and $n \geq 1, m \geq 0$.

Kirkwood (2015) further states that all conditional transition probabilities can be expressed in a transition matrix

$$P = \begin{pmatrix} P_{1,1} & P_{1,2} & \cdots & P_{1,n} \\ p_{2,1} & P_{2,2} & \cdots & P_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n,1} & P_{n,2} & \cdots & P_{n,n} \end{pmatrix} \tag{4.10}$$

where $P_{i,j}$ is the transition probability from state i to state j and n denotes the number of states. Kirkwood (2015) remarks that $P_{i,j} \geq 0$ and

$$\sum_{j=1}^{n} P_{i,j} = 1 \tag{4.11}$$

for all $i = 1, \ldots, n$.

The transition matrix P combined with the probability distribution of the initial condition $P(X_1 = x_1)$ defines the Markov chain model (Hagen et al., 2013).

Anastasiou and Tsekos (1996) states that a maximum likelihood estimator for the transition probabilities for such a process is given by

$$\hat{p}_{ij} = \frac{N_{ij}}{N_i} \tag{4.12}$$

where N_{ij} is the number of observed transitions from state i to state j, and N_i is the total number of occurrences of the state i in the sequence.

An important property of Markov chains for this thesis are absorbing states. Kirkwood (2015) defines an absorbing state as a state where once the process enters that state, it never leaves. This is true if and only if $P_{i,i} = 1$ (Kirkwood, 2015).

Chapter 5

Method

This chapter details the methods used for implementation and validation of the stochastic metocean generators. Three model families were selected for validation:

- Discrete-time finite-state Markov chains
- Vector autoregressive (VAR) and vector autoregressive moving-average (VARMA) models
- Stochastic autoregressive long short-term memory (SAR-LSTM) neural networks

The chapter starts with detailing the observed metocean series used as case study and the transformations applied to the metocean time series before and after simulation. Then, the design decisions and simulation algorithms for each model family is presented. Finally, the validation criteria for quality of results are presented.

5.1 Description and Preprocessing of Observed Time Series

The quality of the models was evaluated based on their ability to replicate specific statistical properties of two different hindcast metocean time series. The statistical properties were chosen based on the use of the synthetic time series in marine simulation-based design and the validation criteria are described in Section 5.6. Only the bivariate time series of significant wave height H_s and peak period T_p was considered in this thesis.

The two metocean time series used as case study were from different hindcast archives, meaning that a different hindcast model was used to generate the hindcast data. See Section 7.1.1 for a discussion on the validity of building metocean generators on hindcast data and a comparison of the different hindcast models.

5.1.1 North Sea Series

The first metocean time series was from an area of the North Sea with coordinates 56.5°N 3.2°E. The location was chosen as it is a commonly used time series at the Department of Marine Technology (IMT) at NTNU.

The time series was provided from the WAM10 hindcast archive and contains wind and sea state parameters from 1958-2016, sampled at three hour intervals (Norwegian Meterological Institute, 2009). Only the observations from year 1990 to 2015 were used in this thesis. The time series contained 75968 observations of H_s and T_p.

Observed significant wave heights in the hindcast data were discretized into 111 unique values, while peak periods were discretized into 23 unique values. The H_s values were linearly spaced, while the T_p values were spaced logarithmic. Due to logarithmic spacing, the difference between the observed values increases with increasing T_P. There were no missing values or apparent errors in the hindcast data.

The T_p values were distributed within the range of the class which it belongs to (i.e., the discrete values) and Andersen (2009) states that it can be assumed that the values are uniformly distributed within its class and randomly distributed by

$$T_{p,distributed} = 3.244e^{0.09525(n-0.5-\epsilon)}, \quad n \in Z \tag{5.1}$$

where ϵ is a random variable uniformly distributed in range [0,1] and

$$n = round(1 + \frac{\ln(\frac{T_P}{3.244})}{0.9525}) \tag{5.2}$$

5.1.2 Mediterranean Sea Series

The second time series was from an area of the Mediterranean Sea with coordinates 41°N 6°E. The location was chosen as it is a commonly used area in the literature concerning metocean simulation (Soares and Cunha, 2000). Further, it was desirable to have an area with calmer metocean conditions and a different time scale than the North Sea series.

The time series was provided from the ERA5 hindcast archive and contains wind and sea state parameters from 1990-2015, sampled at six hour intervals (ECMWF, 2018). The time series contained 37984 observations of H_s and T_p.

The values of H_s were not discretized, but the T_p values were discretized in 372 classes. The classes were inconsistently logarithmic spaced. Several classes were very close to each other and the number of observations in a class varied significantly. No general uniform distribution function such as equation 5.1 could therefore be applied.

In order to distribute the T_p classes, the classes with the highest amount of observations, which were very close to another large class, were merged such that the location of the new class was at the middle point between the two classes. The classes were then distributed by an uniform distribution weighted by the number of observations in the class before and after the current class. The same approach was then applied to classes with a lower number of observations. Different values for the number of observations in large and small classes, in addition to the distance between them, were evaluated until a satisfactory distribution was achieved.

5.2 Data Transformations

The goal of the transformation was to produce stationary Gaussian residuals $\boldsymbol{W}$. The data transformation was similar for the VAR, VARMA and SAR-LSTM models, while the Markov models differs in that it only transforms H_s before simulation and has no requirements for Gaussian distribution. Therefore, the data transformation for VAR, VARMA and SAR-LSTM will be presented first, followed by the transformation used for the Markov models.

The data was first transformed and the resulting residuals were simulated using one of the model families before being back-transformed. The back-transform was performed by applying the inverse of the transformations in the reverse order.

5.2.1 VAR, VARMA and SAR-LSTM Network Transformation

The VAR, VARMA and SAR-LSTM models require that the series is stationary. Both metocean series have a yearly statistical periodicity induced by the annual meteorological cycle, which must be removed before simulation. Further, VAR and VARMA assumes that the stationary process is approximately Gaussian. It was assumed that the SAR-LSTM models generalization would be improved by the same process.

This section follows the order of operations for producing the stationary Gaussian residuals. First, a lognormal Rosenblatt transformation to Gaussian space is presented, followed by the seasonal transformation to stationarity.

Lognormal Rosenblatt transform

The traditional approach for transforming a time series to approximately Gaussian in metocean simulation is to apply the logarithmic transformation or the multivariate Box-Cox transformation (Stefanakos and Belibassakis, 2005; Box and Cox, 1964; Monbet et al., 2007). This thesis introduces the use of a lognormal Rosenblatt transformation for producing approximately Gaussian residuals (Rosenblatt, 1952). The goal of the lognormal Rosenblatt transformation was to improve the model's ability to maintain dependencies throughout simulation. Previous test performed by the author had shown that the Box-Cox transformation resulted in a weak replication of the observed joint distribution.

The lognormal Rosenblatt transformation is based the Conditional Modeling Approach (CMA) as given in Det Norske Veritas (2010). In CMA, the joint density function of H_s and T_p is given by a marginal distribution and a conditional density function (Det Norske Veritas, 2010). The lognormal Rosenblatt transformation deviates from the CMA method by changing the assumptions that H_s follows a 3-parameter Weibull distribution to that it follows a lognormal distribution.

First, the significant wave height H_s was modeled by a lognormal cumulative distribution function

$$F_{H_s}(h) = \Phi\left(\frac{(ln(h) - \mu_{H_s})}{\sigma_{H_s}}\right) \tag{5.3}$$

where μ_{H_s} and σ_{H_s} are the parameters of the lognormal distribution estimated from the observed data and h is the significant wave height H_s. $\Phi(z)$ is the cumulative distribution function of the standard normal distribution.

Next, the peak period T_p was modeled by a lognormal cumulative distribution function conditioned on H_s

$$F_{T_p|H_s}(t|h) = \Phi(\frac{(ln(t) - \mu_{T_p|H_s})}{\sigma_{T_p|H_s}}) \tag{5.4}$$

where t is the peak period T_p. The conditional mean $\mu_{T_p|H_s}$ and conditional variance $\sigma_{T_p|H_s}$ are given as functions of H_s. Det Norske Veritas (2010) states that the following functions often give a good fit for the conditioned lognormal parameters

$$\mu_{T_p|H_s} = \mathbb{E}[lnT_p] = a_0 + a_1 h^{a_2} \tag{5.5}$$

and

$$\sigma_{T_p|H_s} = std[lnT_p] = b_0 + b_1 exp(b_2 h) \tag{5.6}$$

where the parameters a_i and b_i are estimated from the observed data.

The data was then transformed to standard normalized U-space by

$$u_1 = \Phi^{-1}(F_{H_s}(h)) \tag{5.7}$$

$$u_2 = \Phi^{-1}(F_{T_p|H_s}(t|h)) \tag{5.8}$$

The back-transform thus becomes

$$h = \Phi(F_{H_s}^{-1}(u_1)) \tag{5.9}$$

$$t = \Phi(F_{T_p|H_s}^{-1}(u_2)) \tag{5.10}$$

Multivariate Seasonal Transform

The next step was to to make the series stationary by removing the yearly meteorologic periodicity from u_1 and u_2. The seasonal transformation follows the approach set forth in Stefanakos and Belibassakis (2005). It was assumed that the series of approximately Gaussian variables u_1 and u_2 admits the decomposition

$$\boldsymbol{Y}_t = \boldsymbol{M}_t + \boldsymbol{\Sigma}_t \cdot \boldsymbol{W}_t \tag{5.11}$$

where $\boldsymbol{Y}_t$ is a vector of the bivariate series of u_1 and u_2. The mean vector $\boldsymbol{M}_t$ and standard deviation matrix $\boldsymbol{\Sigma}_t$ are deterministic periodic functions with period of one year. Note that in this notation, $\boldsymbol{\Sigma}_t$ denotes the square root of the covariance matrix.

The terms of the seasonal patterns $\boldsymbol{M}_t$ and $\boldsymbol{\Sigma}_t$ were estimated by first reindexing the approximately Gaussian series in terms of $\boldsymbol{Y}_t$ according to the triple notation introduced in Stefanakos and Belibassakis (2005)

$$Y_n(j,k,\tau_k), \qquad n = 1,\ldots,N \tag{5.12}$$

where J is the number of years, k_m is the number of observations within the m^{th} month. The indexes are given by $j = 1,\ldots,J$, $m = 1,\ldots,12$ and $k = 1,\ldots,K_m$. The variable indexes are given by $n,l = 1,2$ for the bivariate series. The time series of monthly mean and covariance can then be determined from

$$M_n(j,m) = \frac{1}{K_m}\sum_{k=1}^{K_m} Y_n(j,m,\tau_k), \quad n = 1,\ldots,N \tag{5.13}$$

and

$$S_{nl}(j,m) = \frac{1}{K_m}\sum_{k=1}^{K_m}\left[Y_n(j,m,\tau_k) - M_{3,n}(j,m)\right]\left[Y_l(j,m,\tau_k) - M_{3,l}(j,m)\right], \quad n,l = 1,\ldots,N \tag{5.14}$$

The mean and covariance of the seasonal patterns can be obtained from

$$\tilde{M}_n(m) = \frac{1}{J}\sum_{j=1}^{J} M_{v,n}(j,m), \quad n = 1,\ldots,N \tag{5.15}$$

and

$$\tilde{S}_{nl}(m) = \frac{1}{J}\sum_{j=1}^{J} S_{v,nl}(j,m), \quad n,l = 1,\ldots,N \tag{5.16}$$

Stefanakos and Athanassoulis (2001) and Stefanakos and Athanassoulis (2003) presented that periodic extensions of $\tilde{M}_n(m)$ and $\tilde{S}_{nl}(m)$ are good estimates of $M_{n,t}$ and $\Sigma_{nl,t}$, respectively. The deterministic periodic extensions $M_{n,t}$ and $\Sigma_{nl,t}$ were then found by fitting Fourier series with period of one year to the inner-annual seasonal patterns $\tilde{M}_n(m)$ and $\tilde{S}_{nl}(m)$.

The residual component $\boldsymbol{W}_t$ was then assumed stationary and approximately Gaussian, given by

$$\boldsymbol{W}_t = \frac{\boldsymbol{Y}_t - \boldsymbol{M}_t}{\boldsymbol{\Sigma}_t} \tag{5.17}$$

Soares and Cunha (2000) states that for some metocean series the seasonal transform

$$\boldsymbol{W}_t = \boldsymbol{Y}_t - \boldsymbol{M}_t \tag{5.18}$$

yields better results.

To avoid any confusion, the seasonal transformations in Equation 5.17 and 5.18 will be denoted as the full seasonal transform and mean seasonal transform, respectively.

Both forms of the seasonal transform were evaluated for the VAR and VARMA models, while only the mean transform was used with the SAR-LSTM models. The lack of access to computational power led to that it was only feasible to evaluate one of the transformations. The mean transform was chosen since it produces less extreme values and is more consistent than the full transform. It was assumed that this would result in the best generalization.

The mean monthly covariance between u_1 and u_2 used for estimating the periodic function Σ_t were found to be very small and in some cases negative. Taking the square root of the covariance matrix then produces complex numbers which must be avoided. It was found that setting the covariances to zero would not significantly influence the results. See Section 7.2.3 for further discussion.

5.2.2 Markov Chain Transformation

The Markov model does not need to fulfill any Gaussian assumptions, but requires that the distribution used to generate the transition matrix $\boldsymbol{P}$ is stationary. Thus, only the seasonal transform to stationarity was applied to H_s. The dependency between the variables was maintained by a coupling matrix that did not require that T_p was stationary or Gaussian.

The seasonal transform for the Markov model was thus in the univariate case and the decomposition assumed to be

$$Y_t = M_t + S_t \cdot W_t \tag{5.19}$$

where Y_t is the H_s time series. The determination of the deterministic periodic function for mean M_t and standard deviation S_t were found by the same approach as the multivariate case above. The transform to stationary residuals W thus becomes

$$W_t = \frac{Y_t - M_t}{S_t} \tag{5.20}$$

and the mean only transform

$$W_t = Y_t - M_t \tag{5.21}$$

was also evaluated due to the statements in Soares and Cunha (2000).

5.3 Markov Chain Stochastic Generator

The Markov chain models are based on the models from the preceding project thesis and follows the same approach, except that Fourier approximation functions are applied instead of a seasonal transform with constant values for each month. A Markov chain is not inherently able to recreate dependencies between parameters and as such requires some method for maintaining the dependencies throughout a simulation. Several different methods are available such as correlation matrices, splines or other coupling methods (Hagen et al., 2013).

A coupling matrix was found to be a good alternative since H_s and T_p were discretized into relatively few values. The coupling matrix $\boldsymbol{C}$ was given by

$$\boldsymbol{C} = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,N_T} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,N_T} \\ \vdots & \vdots & \ddots & \vdots \\ c_{N_H,1} & c_{N_H,2} & \cdots & c_{N_H,N_T} \end{pmatrix} \tag{5.22}$$

where N_T and N_H are the number of unique discrete values of peak period and significant wave height, respectively. C_{ij} denotes the probability of the peak period being in state j when the significant wave height is in state i.

Anastasiou and Tsekos (1996) states that a maximum likelihood estimator for such a process is given by

$$\hat{c}_{ij} = \frac{N_{ij}}{N_i} \tag{5.23}$$

where N_{ij} is the total number of observed state j when in state i, and N_i is the total number of occurrences of state i. This estimator was then used to determine the coupling probabilities.

The distribution functions in Section 5.1 were applied to the peak periods after coupling. The number of unique values in the Mediterranean Sea series was much higher than in the North Sea series, and thus the H_s values in the Mediterranean Sea were rounded to one decimal place when estimating $\boldsymbol{C}$. This was done to reduce the size of $\boldsymbol{C}$ such that the sampling from this distribution would be more realistic. It also avoids problems related to absorbing states.

Algorithm 3 Markov Simulation Algorithm

Require: Time series $\boldsymbol{Y}$
Require: Number of replications N
  Build Coupling matrix: C
  Apply transformation: $W_t = \frac{H_{s,t} - M_t}{S_t}$ or $W_t = H_{s,t} - M_t$
  Build transition matrix: P
  Sample initial condition $\hat{W}_0$ from W_t distribution
  Initialize time step: $t = 1$
  while $t \leq N$ **do**
    Sample: $\hat{W}_t = P(\hat{W}_{t-1})$
    Back-transform: $\hat{H}_{s,t} = M_t + \hat{W}_t$ or $\hat{H}_{s,t} = M_t + S_t\hat{W}_t$
    Sample: $\hat{T}_{p,t} = C(\hat{H}_{s,t})$
    $t \leftarrow t + 1$
  end while

The number of possible states in the transition matrix $\boldsymbol{P}$ is a parameter that must be determined before simulation. It was assumed that increasing the number of states would result in a model of a higher quality. This was shown to be the case for the Markov chain models in the project thesis. Thus, the number of states was found by finding the maximum number of states before

absorbing states appeared. The simulation method is summarized in Algorithm 3 as a pseudo-alogrithm.

5.4 VAR and VARMA Stochastic Generator

The data was first transformed by the lognormal Rosenblatt transformation and the seasonal transformation as detailed in Section 5.2.1. Let $trans(\boldsymbol{Y}) = \boldsymbol{W}$ denote the complete transformation from environmental space $\boldsymbol{Y}$ to residual space $\boldsymbol{W}$. The residuals $\boldsymbol{W}_t$ were assumed stationary and Gaussian, which means that an appropriate VAR or VARMA model could be fitted to the stochastic process. In addition to determining the parameters $\boldsymbol{c}$, $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$, the optimal number of lags p and q for the AR and MA component, respectively, must also be determined.

Based on the results of Stefanakos and Belibassakis (2005), VARMA(p,q) models with lag values $p,q = 1,\ldots,4$ were evaluated. Pure VAR(p) models with lags $p = 1,\ldots,30$ were also evaluated. Higher order VAR(p) models showed comparable results to VARMA(p,q) in Soares and Cunha (2000). The number of evaluated lags were much higher for the VAR(p) models since it is significantly less computationally expensive to determine the parameters of a VAR(p) model compared to a VARMA(p,q) model (Wei, 1990).

The parameters of the VAR models were found by maximum likelihood estimation (MLE) (Wei, 1990). VAR models for all lag values were estimated by MLE and the optimal number of lags p^* was found by Akaike information criterion (AIC). Akaike (1973) defines AIC as

$$AIC = 2k - 2ln(\hat{L}) \tag{5.24}$$

where k is the number of estimated parameters and $\hat{L}$ is the maximum value of the likelihood function. AIC is commonly used for selecting a model from a set of candidates, where the model with the lowest AIC minimizes the information loss (Wei, 1990). Brockwell and Davis (2009) recommends using AIC to find the optimal number of lags for both VAR(p) and VARMA(p,q). The optimal number of lags was then found as $p^* = min(AIC(VAR(p)))$ for $p = 1,\ldots,30$.

The VARMA models followed a different procedure than the VAR models. The determination of number of VARMA(p,q) lags followed the approach of Tiao and Tsay (1983). The two-way table of the P-values of the extended cross-correlation matrices was first computed by the multivariate Ljung-Box statistics of the series (Tsay, 2014). The optimal number of lags were chosen as those that had the lowest P-value at the 5% significance level (Tsay, 2014). AIC could not be used as it was too time consuming, due to available computational power, to estimate the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ for each possible combination of p and q. The parameters of the model with the optimal lags p,q was then estimated by maximum likelihood (Wei, 1990).

The process could now be simulated using either of the model types. It is common to let the simulation run for some amount of time steps from the initial conditions to remove the influence of the initial conditions (Martin et al., 2012). This is called the burn-in phase of the simulation. Both models used burn-in with a length of 400 time steps.

The pseudo-algorithms for simulation with VAR and VARMA models are summarized in Algorithm 4 and 5, respectively.

Algorithm 4 VAR Simulation Algorithm

Require: Time series $\boldsymbol{Y}$, maximum number of lags to be evaluated P
Require: Number of replications N, burn-in length B
  Apply transformation: $\boldsymbol{W} = trans(\boldsymbol{Y})$
  Estimate parameters of VAR(p) for $p = 1, \ldots, P$ by MLE
  Select optimal lag: $p^* = min(AIC(VAR(p)))$
  Initialize time step: $t = 1$
  Initialize initial condition: $\boldsymbol{L} = \boldsymbol{W}_1, \ldots, \boldsymbol{W}_{p^*}$
  while $t \leq N + B$ **do**
    Sample: $\hat{\boldsymbol{W}}_t = VAR(p^*, \boldsymbol{L})$
    Update lookback: $\boldsymbol{L}_{\tau+1} = \boldsymbol{L}_\tau$ for $\tau = 1, \ldots, (p^* - 1)$ and $\boldsymbol{L}_1 = \hat{\boldsymbol{W}}_t$
    Back-transform: $\hat{\boldsymbol{Y}}_t = trans^{-1}(\hat{\boldsymbol{W}}_t)$
    $t \leftarrow t + 1$
  end while
  Drop $\hat{\boldsymbol{Y}}_1, \ldots, \hat{\boldsymbol{Y}}_B$ from $\hat{\boldsymbol{Y}}$

Algorithm 5 VARMA Simulation Algorithm

Require: Time series $\boldsymbol{Y}$, maximum number of lags to be evaluated P, Q
Require: Number of replications N, burn-in length B, Significance level α
  Apply transformation: $\boldsymbol{W} = trans(\boldsymbol{Y})$
  Determine P-values of extended crosscorrelation matrix: $\boldsymbol{pECCM}(p, q)$ for $p, q = 1, \ldots, P, Q$
  Determine optimal lags: $p^*, q^* = min(\boldsymbol{pECCM}(p, q) \geq \alpha)$
  Estimate parameters of VARMA(p^*, q^*) by MLE
  Initialize time step: $t = 1$
  Initialize initial condition: $\boldsymbol{L} = \boldsymbol{W}_1, \ldots, \boldsymbol{W}_{p^*}$
  while $t \leq N + B$ **do**
    Sample: $\hat{W}_t = VARMA(p^*, q^*, \boldsymbol{L})$
    Update lookback: $\boldsymbol{L}_{\tau+1} = \boldsymbol{L}_\tau$ for $\tau = 1, \ldots, (p^* - 1)$ and $\boldsymbol{L}_1 = \hat{\boldsymbol{W}}_t$
    Back-transform: $\hat{\boldsymbol{Y}}_t = trans^{-1}(\hat{\boldsymbol{W}}_t)$
    $t \leftarrow t + 1$
  end while
  Drop $\hat{\boldsymbol{Y}}_1, \ldots, \hat{\boldsymbol{Y}}_B$ from $\hat{\boldsymbol{Y}}$

5.5 SAR-LSTM Stochastic Generator

There currently exist no standard method for using deep learning models as stochastic generators. Therefore a new method for using deep learning models as stochastic generators is presented in this thesis, stochastic autoregressive long short-term memory (SAR-LSTM) generators. The general idea for the SAR-LSTM model was to use a prediction network with a white noise component. The SAR-LSTM model is conceptually similar to an AR model, in that there is a deterministic autoregressive component which gives the predicted value given past values and another stochastic white noise component given by the marginal distributions of the process.

Training the SAR-LSTM model required that some performance measure was specified. It was assumed that the deep learning model which had the lowest error in prediction generalization would also be the best stochastic generator. LSTM neural networks were chosen as the model type since they are the state-of-the-art within prediction problems and has shown great results in this area (Goodfellow et al., 2016).

Designing a SAR-LSTM stochastic metocean generator forces the designer to make several design decisions. The problem must first be framed in a way which the algorithm is able to learn. A model family must then be chosen which has it own design decisions for architecture, cost function, optimization method, units, hyperparameters and so on. This section details the assumptions, design decisions and proposed method for metocean simulation with SAR-LSTM network stochastic generators.

5.5.1 Supervised Learning Problem

Transformation and Scaling

The metocean data was first transformed to stationary approximately Gaussian residuals $\boldsymbol{W}$ by the transformation in Section 5.2.1. A LSTM network generally performs better when all features are in the range [0,1] (Brownlee, 2017). A minmax scaler was therefore applied to $\boldsymbol{W}$, which normalizes each feature individually, such that the range of the training set is [0,1] (Chollet, 2015). Additionally, the minmax scaler was applied to the month indexes which were then concatenated to their corresponding position in $\boldsymbol{W}$.

Reshaping Metocean Data to Supervised Learning Problem

The scaled residuals and month indexes were then transformed into a task which the algorithm may be able to learn. The metocean dataset was framed as a supervised learning prediction problem where the goal was to predict H_s and T_p at the current hour t, given some features at prior time steps $t-1, t-2, \ldots$. The features chosen from the metocean data were

- Past observations of H_s
- Past observations of T_p
- Past observations of month number

Each of the features forms a set of past values, where each feature set has equal length. The number of past values used as input was determined by the length of the look-back window L, i.e, how many lags are used for prediction of the next value. The length of the look-back window L was assumed to significantly influence the performance and was therefore chosen as a hyperparameter. A look-back window of features thus forms an example $\boldsymbol{x}$ of the dataset. The current observations of H_s and T_p becomes the labels $\boldsymbol{y}$.

The month was only used as a feature, as it was the monthly variability which was interesting and not the prediction of this value. This was included as a feature to model any remaining monthly statistical properties.

The examples and labels were generated by shifting the series back in time for each time step of the look-back window. The exact shape of the inputs $\boldsymbol{x}$ depends on the machine learning algorithm. In this thesis, the high-level neural networks API Keras was used (Chollet, 2015). TensorFlow, an open source library for for high performance numerical computation, was used as back-end (Abadi et al., 2015). Keras assumes that the examples $\boldsymbol{x}$ for LSTM networks are on the form [samples, timesteps, features] (Chollet, 2015).

Training and Validation Data

Next, the supervised learning problem formulation of the metocean series was split into a training set and a validation set. The training set was used to train the model while the validation set was used to find the best values of the hyperparameters. The data set was split by the 80-20 rule, such that 80 percent was used for training and 20 percent for validation.

There was no need for a test set since the model was to be validated based on its quality as a stochastic generator, not how good it performed in predictions on unseen data. Regardless, the North Sea observations for year 1989, note that these values were not part of the training or validation set, were used as a test set to test the prediction performance.

5.5.2 LSTM Network Architecture and Functions

Deeper LSTM networks have been shown to have improved performance over shallow networks (Pascanu et al., 2013). Therefore, the number of LSTM cells (layers) was selected as another hyperparameter. When LSTM cells are stacked, they return their sequences to the next cell until the final LSTM cell. This cell output the hidden states to an output layer, similar to a feedforward layer, with two output units. Two output units were required since there were two labels. The output units then give the actual prediction and are used by the cost function for training the network.

Hidden and Output Unit Functions

Selection of the hidden and output unit functions for a prediction LSTM network model is simpler than for a feedforward network due to some amount of standardization. A LSTM network

does not have several hidden units, such as a feedforward network, but it has a similar hyperparameter which is the number of hidden states in a LSTM cell. The hidden states in LSTM networks are often referred to as units in a lot of the literature and documentation, but there is only one unit in each cell, i.e, the cell itself (Goodfellow et al., 2016)(Chollet, 2015). The number of hidden states have a major influence on the performance and was therefore selected as another hyperparameter.

The hidden unit functions were kept at the default setting with sigmoid function controlled gates and $tanh(z)$ as the output function of the cells. The final output units had linear output unit functions, since it is preferred for real-valued outputs such as time series (Brownlee, 2017).

Cost Function and Optimization Algorithm

A good choice for cost function becomes the root mean square error (RMSE) when framed as a supervised prediction problem. This is the most commonly used cost function for prediction problems (Brownlee, 2017). Further, RMSE works well with linear output units (Goodfellow et al., 2016)

Next, an optimization algorithm which works with the RMSE function and linear output units must be chosen. Adam was chosen as it has been shown to be quite robust and effective with its default settings. It was desirable to have an optimization algorithm which did not require the learning rate to be a hyperparameter of the model. Further, Adam works well with RMSE cost function and linear output units (Kingma and Ba, 2014).

The stochastic mini-batch method was used to find the expectation of the gradient and the training and validation data were not shuffled during sampling to preserve the temporal dependence. The batch size was set to 64 examples for each iteration. This value was found by taking the minimum multiple of two, larger than 32, which led to a practical training time.

Regularization Methods

Three types of regularization methods were used to reduce the generalization error. Dropout was applied with a dropout probability of 50 percent. L^2 regularization is commonly used in prediction models and the weight α was chosen as hyperparameter since the performance of the model may be very sensitive to this weight. Finally, early stopping was implemented by simply saving the parameters of the model each time the generalization error improved.

5.5.3 Grid Search for Optimal Hyperparameters

The LSTM model was now specified and parameterized by four hyperparameters. A grid search was then performed to find good values for the chosen hyperparameters. Generally, a grid search involves picking hyperparameter values approximately on a logarithmic scale or as multiples of two (Goodfellow et al., 2016). The grid search then consists of training a network for

each combination in the hyperparameter sets and measuring some performance criteria (Goodfellow et al., 2016). The RMSE validation loss was chosen as the performance criteria and the best performing model was the model which achieved the lowest validation loss over the training epochs.

The sets of chosen hyperparameters for grid search can be seen Table 6.1.

Table 5.1: Hyperparameters for Grid Search

Hyperparameter	Set
No. cells	{1,2,3}
No. hidden states	{32,64,128}
L^2 weight α	$\{0, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$
Look-back window length	{8,16,32}

A grid search is often repeated at a higher fidelity, i.e., generating a new set of grid values closer to the current best hyperparameter values (Goodfellow et al., 2016). Only a single grid search was performed in this thesis due to time and resource constraints. The grid search was run for 60 epochs with resetting of the model states between each epoch. More epochs and larger sets would have been desirable, but was not possible due to time and resource limitations.

5.5.4 Transforming a Prediction Model to a Simulation Model

After determining the best values for the hyperparameters, a new model with the same parameters was defined, and then trained for 500 epochs. This was to attempt to further decrease the generalization error. The deterministic LSTM component of the model had then been determined and the random white noise component was introduced to form a simulation model.

The distribution of the transformed unscaled variables $\boldsymbol{W}$ were approximately Gaussian and therefore a Gaussian distribution was fitted to the residuals of significant wave height $\boldsymbol{W}_1$ and peak period $\boldsymbol{W}_2$. Let $X_1 \sim \mathcal{N}(0, \sigma_{\boldsymbol{W}_1})$ and $X_2 \sim \mathcal{N}(0, \sigma_{\boldsymbol{W}_2})$ denote the zero-mean white noise Gaussian processes of H_s and T_p respectively, where $\sigma_{\boldsymbol{W}_1}$ and $\sigma_{\boldsymbol{W}_2}$ are the variances of Gaussian distributions estimated from the residuals. After the network made a prediction, the inverse normalization scaling was applied to the predictions. Then, the zero-mean white noise

$$\boldsymbol{X} = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \tag{5.25}$$

was added to the predictions before applying the normalization scaler again. The predications were then added to the look-back window as the observation (sample) at timestep $t-1$. Finally, the predictions were transformed to the environmental space by the inverse transformation $trans^{-1}(\boldsymbol{W}) = \boldsymbol{Y}$ as given in Section 5.2.1.

The complexity of LSTM networks makes it difficult to write a comprehensive algorithm for simulation. The pseudo-algorithm below is intended more as a summary of how the SAR-LSTM network simulation was performed.

Algorithm 6 SAR-LSTM Network Simulation Algorithm

Require: Time series $\boldsymbol{Y}$, Number of replications N, burn-in length B
Require: Set of hyperparameters: $\boldsymbol{H}$
  Apply transformation: $\boldsymbol{W} = trans(\boldsymbol{Y})$
  Scale each feature in $\boldsymbol{W}$ to range [0,1]
  Reshape $\boldsymbol{W}$ into supervised learning problem $\boldsymbol{W}_{supervised}$
  Split $\boldsymbol{W}_{supervised}$ into training set $\boldsymbol{x}_{train}, \boldsymbol{y}_{train}$ and validation set $\boldsymbol{x}_{val}, \boldsymbol{y}_{val}$
  Grid search for optimal hyperparameters $\boldsymbol{H}^*$
  Initialize time step: $t = 1$
  Initialize look-back initial condition: $\boldsymbol{L} = \boldsymbol{W}_{supervised,1}, \ldots, \boldsymbol{W}_{supervised,\boldsymbol{H}^*_{look-back}}$
  while $t \leq N + B$ **do**
    Sample: $\hat{\boldsymbol{W}}_t = LSTM(\boldsymbol{L})$
    Inverse scale $\hat{\boldsymbol{W}}_t$
    Add white noise: $\hat{\boldsymbol{W}}_t \leftarrow \hat{\boldsymbol{W}}_t + \boldsymbol{X}$
    Back-transform: $\hat{\boldsymbol{Y}}_t = trans^{-1}(\hat{\boldsymbol{W}}_t)$
    Scale $\hat{\boldsymbol{W}}_t$
    Update lookback: $\boldsymbol{L}_{\tau+1} = \boldsymbol{L}_\tau$ for $\tau = 1, \ldots, (\boldsymbol{H}^*_{look-back} - 1)$ and $\boldsymbol{L}_1 = \hat{\boldsymbol{W}}_t$
    $t \leftarrow t + 1$
  end while
  Drop $\hat{Y}_1, \ldots, \hat{Y}_B$ from $\hat{Y}$

5.6 Validation

The models quality of results were validated based on their ability to match the following statistical properties of the observed time series:

1. Overall and monthly mean, variance and covariance
2. Joint distribution and marginal distributions
3. Auto- and crosscorrelation
4. Persistence of H_s above or below thresholds (access and waiting windows)

These statistical properties were found to be of the highest importance for applications in simulation-based design. More complex persistence properties such as the joint persistence was omitted due to the persistence of H_s being sufficient for comparison of the models. Validation of the persistence criterion was adapted from the approach presented by Walker et al. (2013). Monte Carlo simulation was applied to approximate the statistical properties and each model type was validated on both the North Sea and Mediterranean Sea series.

Chapter 6

Results

This chapter presents the validation results of the best performing stochastic generators for both metocean time series under consideration. Additional results related to the transformations used and the SAR-LSTM network performance are also presented.

The results of the Markov chain models were of much lower quality than the VAR and VARMA models, and are therefore not presented in this chapter. Additionally, the Markov models with full seasonal transform produced results of such low quality that they were omitted from this thesis. Appendix B contains the results for every model implemented for both the North Sea and Mediterranean Sea series. The scripts used for implementation, simulation and visualization can be found in Appendix C.

The selection of the best performing models depends on the end-user's ranking of the performance criteria. No single VAR or VARMA model performed best in all of the criteria for each series. Thus, the models chosen for comparison against the SAR-LSTM network models were chosen as the models which performed best overall.

VARMA(2,3)$_f$ performed best or was equal to the other VAR and VARMA models in most criteria for the North Sea series, except for autocorrelation of T_p and crosscorrelation, where it slightly underperformed. VARMA(2,3)$_f$ was therefore chosen as the comparison model for the North Sea series. The subscript f and m denotes the type of seasonal transformation to normality used, where f is the full seasonal transform and m is the mean seasonal transform.

VAR(10)$_f$ performed best or was equal to the other VAR and VARMA models in all criteria for the Mediterranean Sea series, except for persistence below threshold where it slightly underperformed. VAR(10)$_f$ was therefore chosen as the comparison model for the Mediterranean Sea series.

6.1 Grid Search Results

Table 6.1 shows the results of the hyperparameter grid search for both series. The LSTM networks component of the SAR-LSTM model thus have the hyperparameters detailed in Table 6.1, in addition to the parameters and functions determined in Section 5.5.

Table 6.1: Hyperparameters for LSTM Networks

Hyperparameter	North Sea	Mediterranean Sea
No. cells	2	2
No. hidden states	128	128
L^2 weight α	0	0
Look-back window length	32	16

6.2 Validation Results

6.2.1 Lower Order Statistical Moments

Overall Lower Order Statistical Moments

Table 6.2 and 6.3 depicts the overall lower order statistical moments of the models for the North Sea series and Mediterranean Sea series, respectively. The overall lower order statistical moments were found as the value of the moments calculated over the entire series.

Table 6.2: Overall Lower Order Statistical Moments for North Sea Series

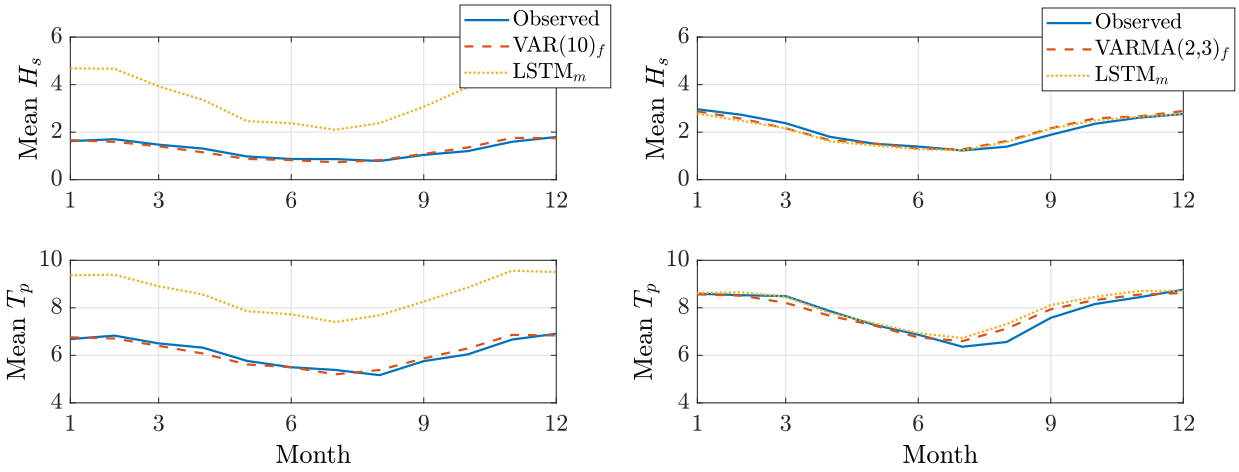
Series	Mean H_s	Variance H_s	Mean T_p	Variance T_p	Covariance
Observed	2.081	1.688	7.786	4.864	1.494
LSTM$_m$	2.050	7.723	7.984	13.321	5.724
VARMA(2,3)$_f$	2.106	2.047	7.840	5.003	1.708

Table 6.3: Overall Lower Order Statistical Moments for Mediterranean Sea Series

Series	Mean H_s	Variance H_s	Mean T_p	Variance T_p	Covariance
Observed	1.267	0.999	6.122	2.947	1.414
LSTM$_m$	3.565	18.626	8.584	9.175	10.360
VAR(10)$_f$	1.249	1.150	6.121	2.947	1.409

Monthly Lower Order Statistical Moments

Figure 6.1 depicts a comparison plot of the monthly mean value of H_s and T_p for both series. The monthly mean was found as the mean of all values within a given month.

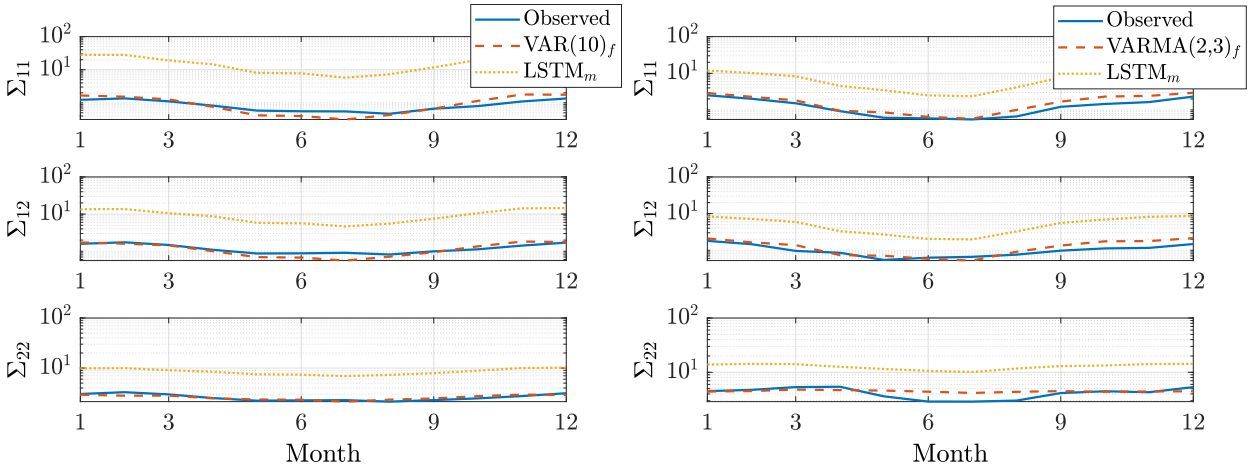


(a) Mediterranean Sea Series

(b) North Sea Series

Figure 6.1: Validation Plot of Monthly Mean of H_s and T_p

Figure 6.2 depicts a comparison plot of the monthly variances and covariance for both series. It is important to note that the y-axis is logarithmic. This was necessary to fit and make both series comparable in a single plot, due to the large differences in values. The y-axis label further denotes the position of the property in the covariance matrix. The monthly variance and covariance were found as the mean of the variance or covariance for a given month for the entire series.



(a) Mediterranean Sea Series

(b) North Sea Series

Figure 6.2: Validation Plot of Monthly Variance and Covariance of H_s and T_p

6.2.2 Joint and Marginal Distributions

Figure 6.3 depicts the scatter plots of the models superimposed on the observed values for the Mediterranean Sea series. Superimposing both models on the observed values made it difficult to simultaneously determine the goodness of the joint distribution and it was therefore split into the two plots seen in Figure 6.3. The left figure depicts the SAR-LSTM network model, while the right figure depicts VAR(10)$_f$.

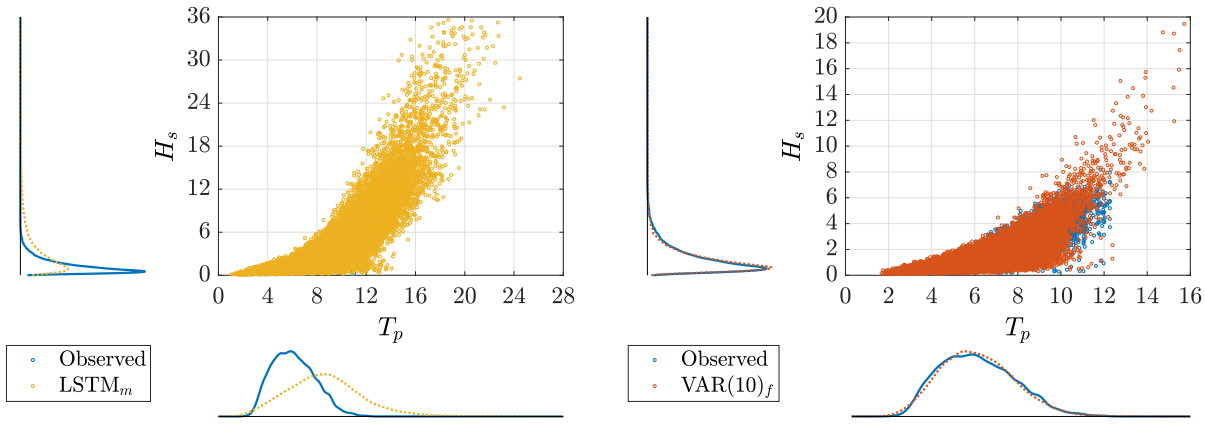
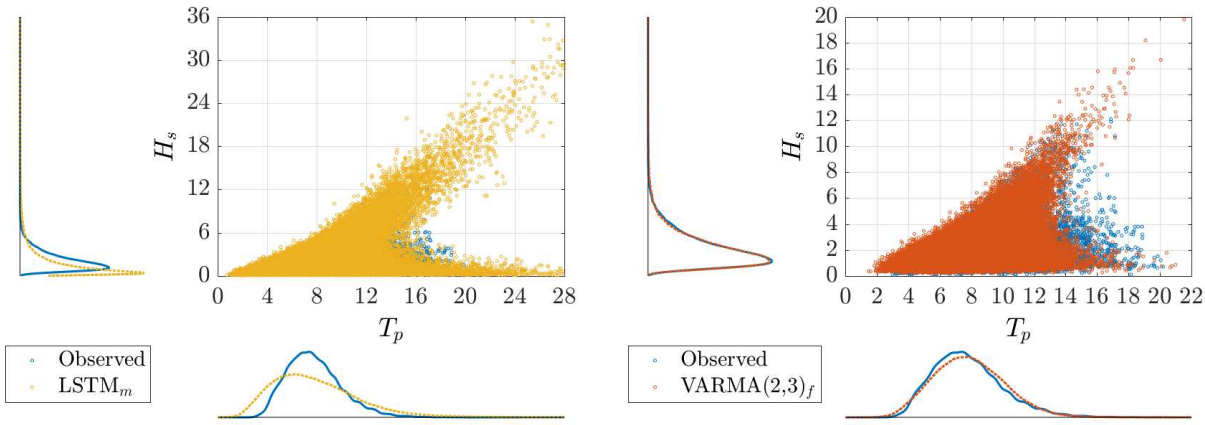


Figure 6.3: Validation Scatter Plot of H_s and T_p for Mediterranean Sea Series

Figure 6.4 depicts the scatter plots of the models superimposed on the observed values for the North Sea series.



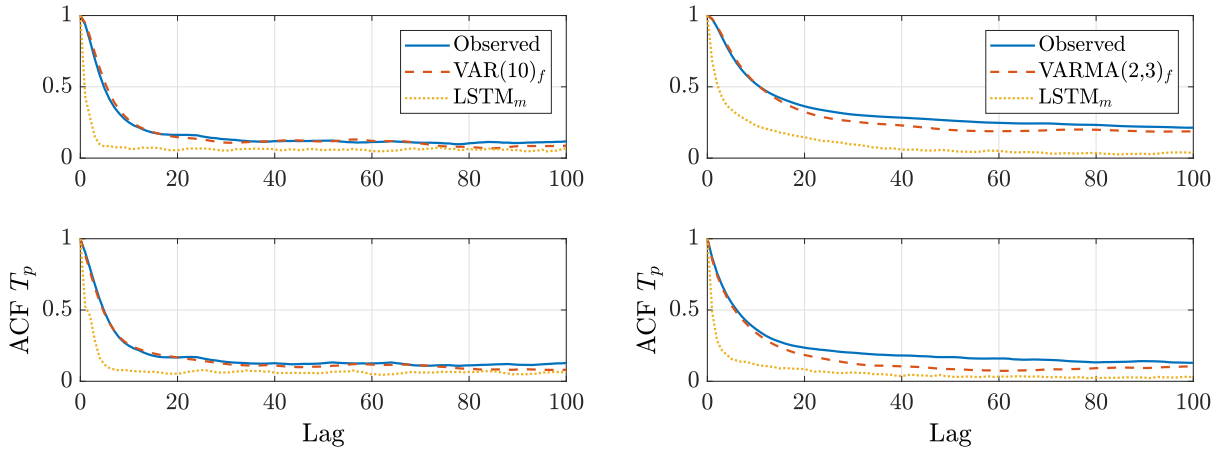
(a) Mediterranean Sea Series

(b) North Sea Series

Figure 6.4: Validation Scatter Plot of H_s and T_p for North Sea Series

6.2.3 Auto- and Crosscorrelation

Figure 6.5 depicts a comparison plot of the H_s and T_p autocorrelation function for both series.

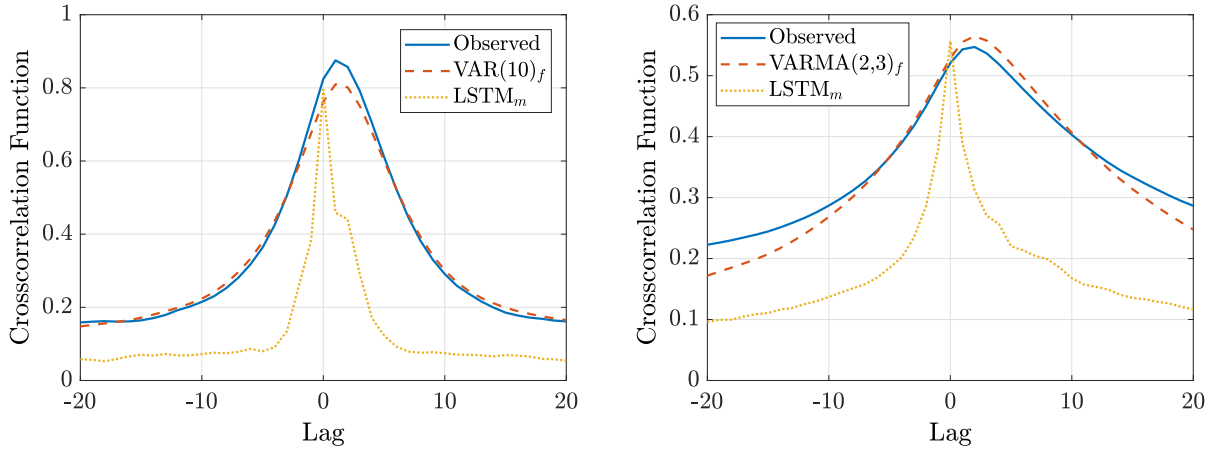


(a) Mediterranean Sea Series

(b) North Sea Series

Figure 6.5: Validation Plot of H_s and T_p Autocorrelation Function

Figure 6.6 depicts a comparison plot of the H_s and T_p crosscorrelation function for both series.



(a) Mediterranean Sea Series

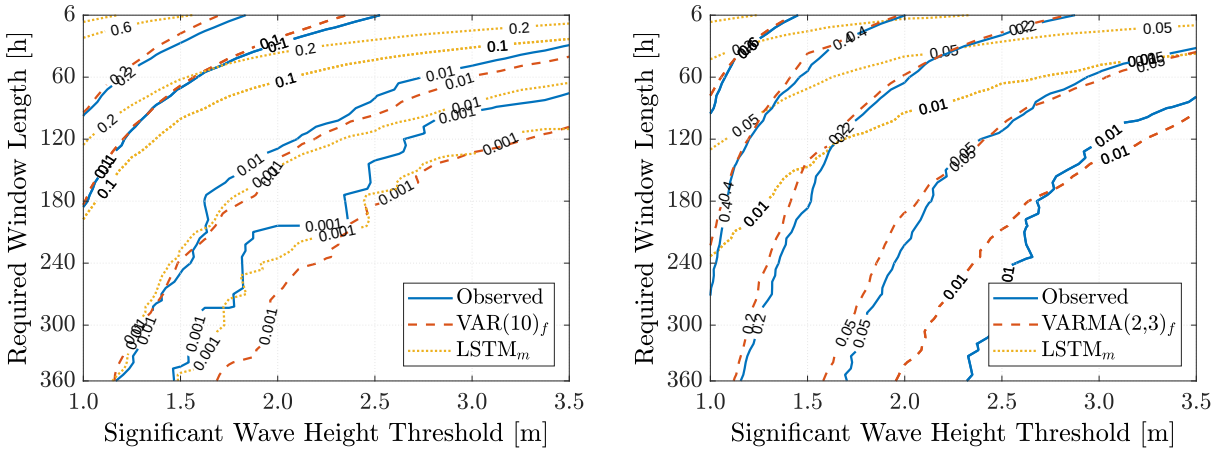
(b) North Sea Series

Figure 6.6: Validation Plot of H_s and T_p Crosscorrelation Function

6.2.4 Persistence

The persistence of the time series was calculated as the number of occurrences of a specific weather window length below or above a threshold defined by the significant wave height. The persistence tables were normalized by calculating the cumulative duration in hours of each cell and then dividing by the total duration of the series. This was done to make the persistence of the models comparable to each other and the observed values.

Figure 6.7 depicts the normalized contours for persistence *above* significant wave height thresholds for both series.

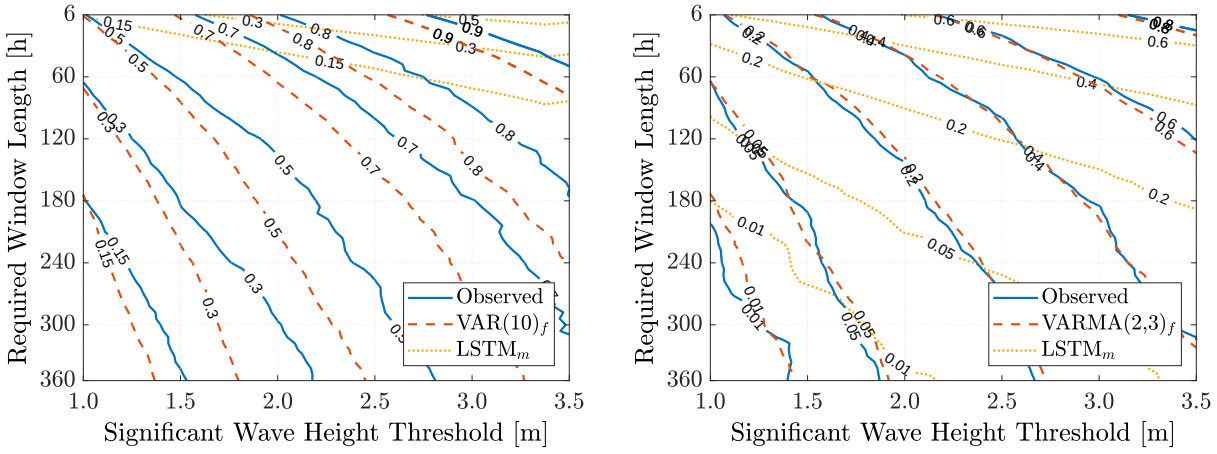


(a) Mediterranean Sea Series

(b) North Sea Series

Figure 6.7: Validation Plot of Normalized Persistence Contours Above Thresholds

Figure 6.8 depicts the normalized contours for persistence *below* significant wave height thresholds for both series.



(a) Mediterranean Sea Series

(b) North Sea Series

Figure 6.8: Validation Plot of Normalized Persistence Contours Below Thresholds

6.3 Prediction and Loss Performance of LSTM Network Models

Figure 6.9 depicts the prediction performance of the LSTM network component of the SAR-LSTM model for the residuals $\boldsymbol{W}$ on a part of the North Sea test set, as described in Section 5.5.1. The root mean square error for each of the residuals over the entire test set is given Table 6.4.

Table 6.4: Root Square Mean Error (RMSE) for North Sea Test Set Predictions

Prediction Output	RMSE
W_1 (H_s)	0.19
W_2 (T_p)	0.60

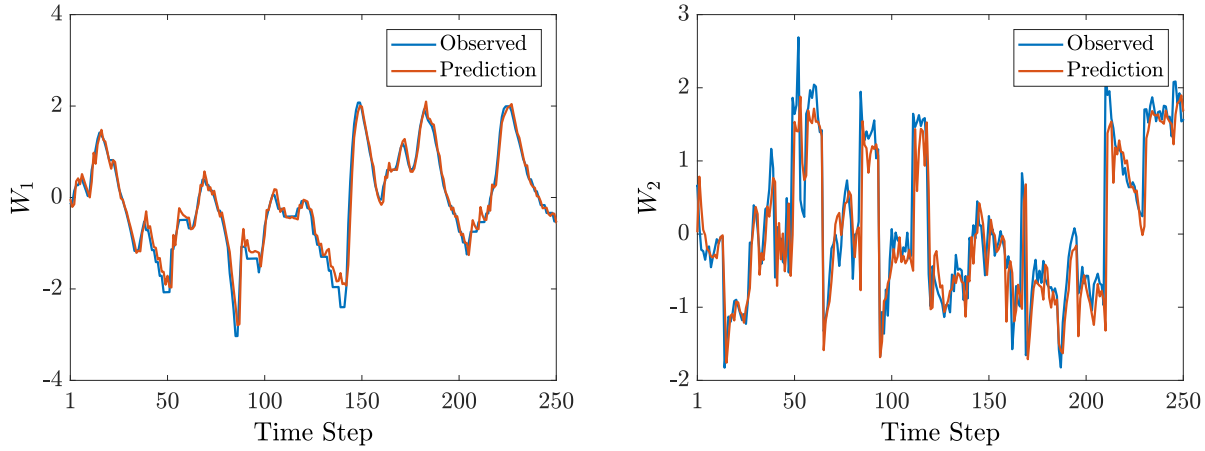
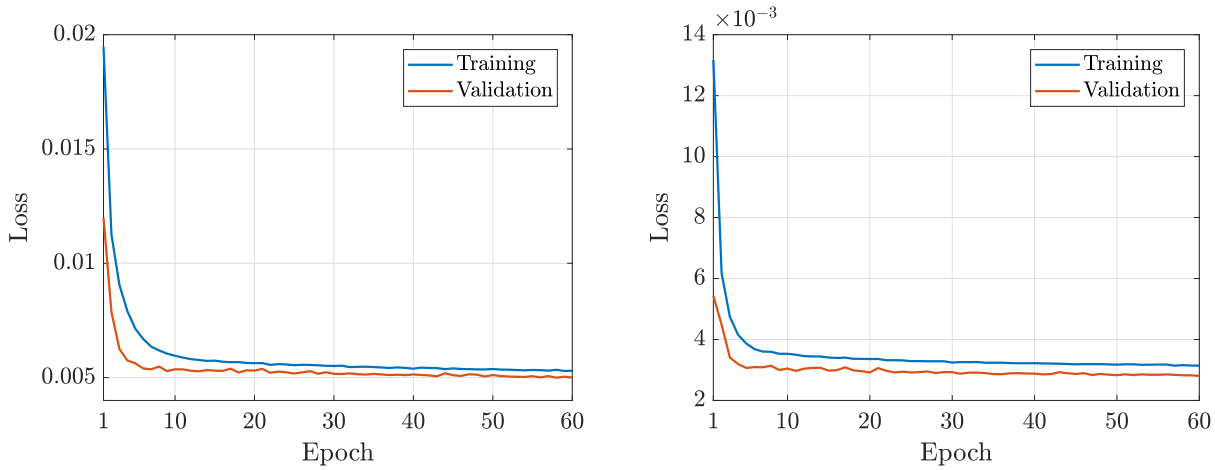


Figure 6.9: Plot of LSTM Predictions on North Sea Test Set

Figure 6.10 depicts the development of the training and validation loss (error) for each LSTM network for the first 60 epochs.



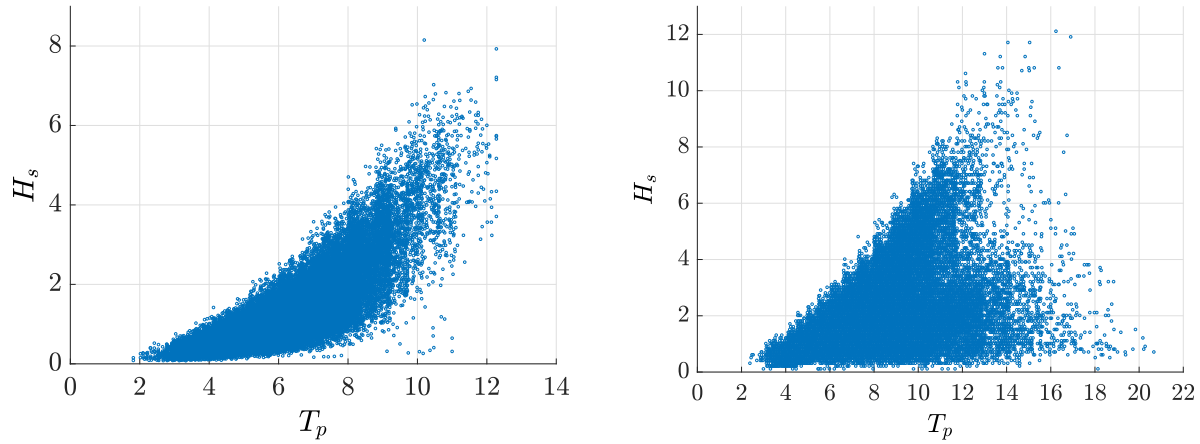
(a) Mediterranean Sea

(b) North Sea

Figure 6.10: Plot of Training and Validation Loss for LSTM Models

6.4 Distributed Observed Scatter Plots

Figure 6.11 depicts the scatter plots of the observed series after applying the corresponding distribution method for T_p, as detailed in Section 5.1.



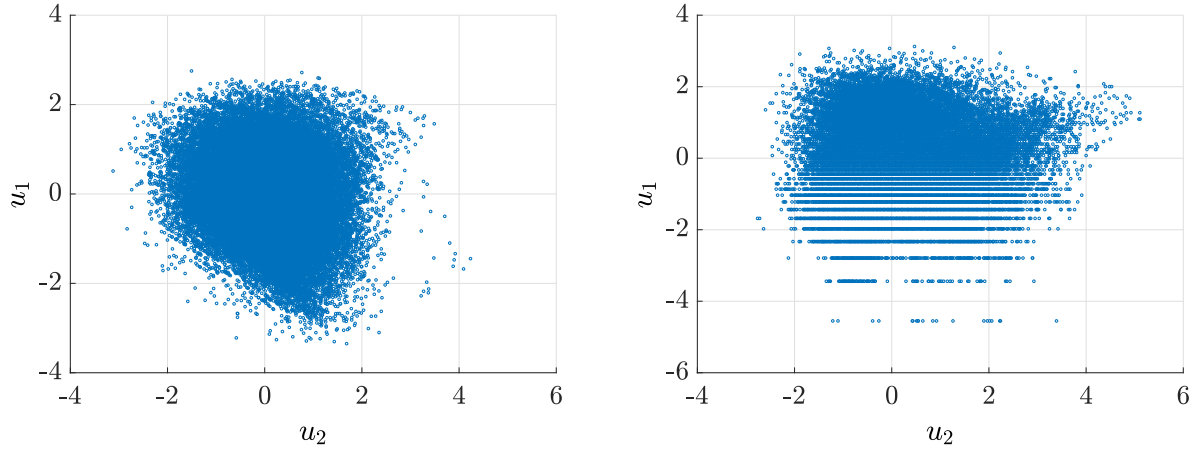
(a) Mediterranean Sea

(b) North Sea

Figure 6.11: Scatter Plot of Distributed T_p and H_s for each Observed Series

6.5 Scatter Plots and Distributions of Residuals

Figure 6.12 depicts the scatter plots of the residuals of the lognormal Rosenblatt transformation, u_2 and u_1, for the observed series.

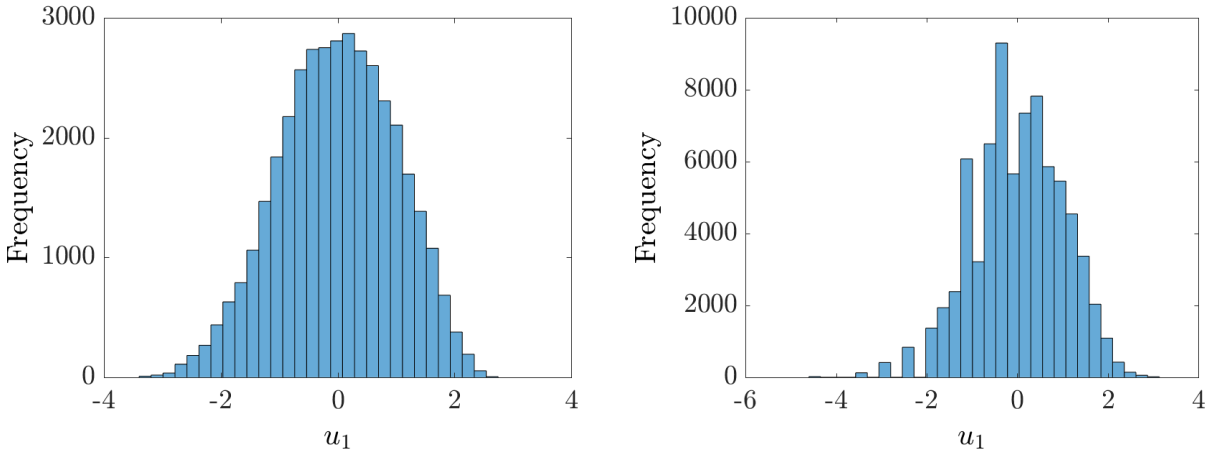


(a) Mediterranean Sea

(b) North Sea

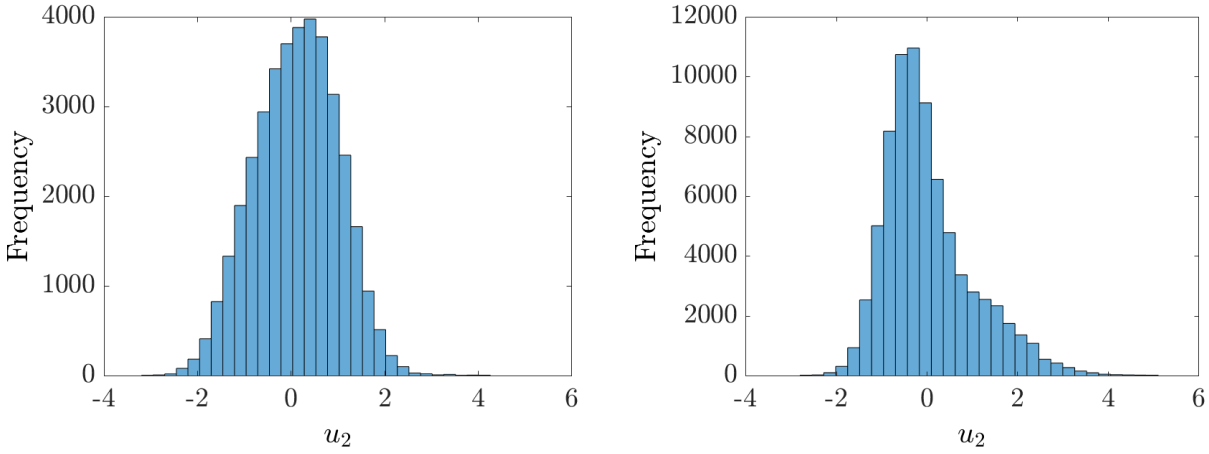
Figure 6.12: Scatter Plot of u_2 and u_1 for each Observed Series

Figure 6.13 and 6.14 depicts the histograms for the residuals u_1 and u_2 of the lognormal Rosenblatt transformation for the observed series.



(a) Mediterranean Sea (b) North Sea

Figure 6.13: Histogram of u_1 for each Observed Series

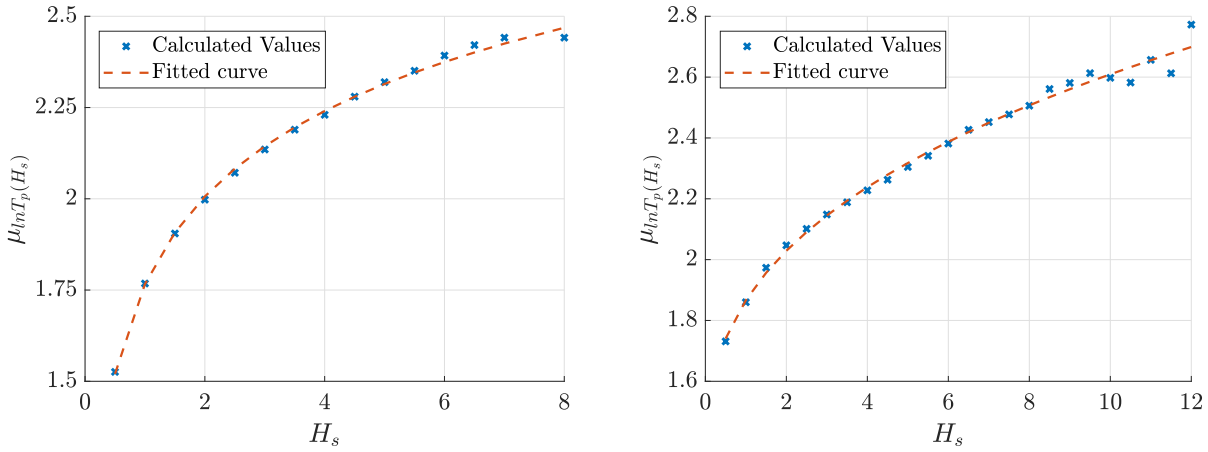


(a) Mediterranean Sea (b) North Sea

Figure 6.14: Histogram of u_2 for each Observed Series

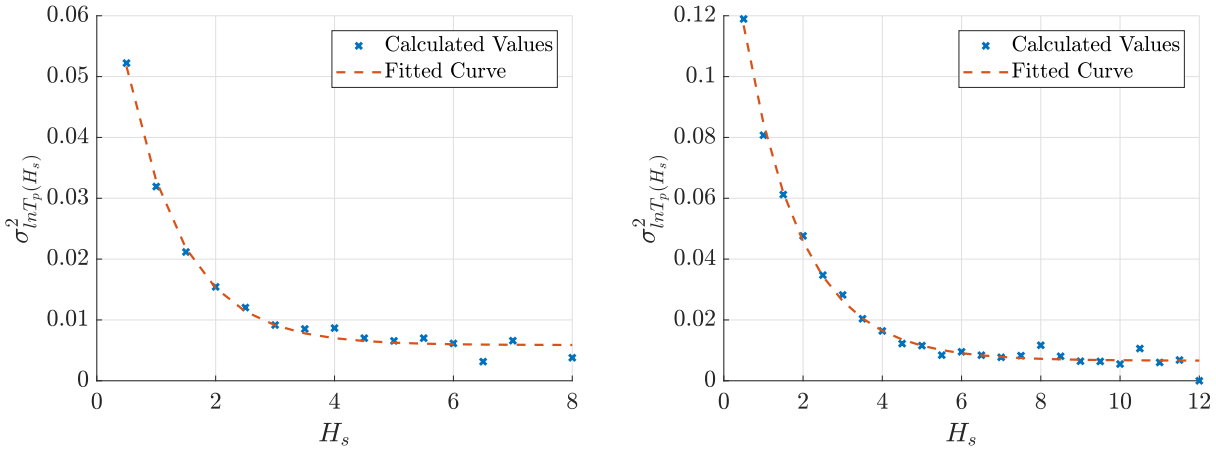
6.6 Curve Fit for Lognormal Conditional Parameters

Figure 6.15 and 6.16 depicts the fitted curves for determining the conditional mean and variance parameters of the lognormal distribution for T_p conditioned on H_s.



(a) Mediterranean Sea (b) North Sea

Figure 6.15: Curve Fit for Conditional Mean of $ln(T_p)$ Given H_s



(a) Mediterranean Sea (b) North Sea

Figure 6.16: Curve Fit for Conditional Variance of $ln(T_p)$ Given H_s

Chapter 7

Discussion

7.1 Metocean Series

7.1.1 Hindcast Data: Building a Model on a Model

It is important to note that the hindcast data which was used for implementation and validation of the models the result of a numerical model. The different hindcast archives used in this thesis were provided from the WAM10 hindcast archive (Norwegian Meterological Institute, 2009) and ERA5 hindcast archive (ECMWF, 2018).

The wave conditions in both models are found from the WAve Model (WAM) model. Komen et al. (1996) provides an in-depth explanation of the WAM model, including scientific basis, actual implementation and applications. Campos and Soares (2016) compared and evaluated three wave hindcast models in the North Atlantic Ocean: ERA-Interim, NOAA/CFSR and HIPOCAS. Both HIPOCAS and ERA-Interim are based on the WAM model. Campos and Soares (2016) found that the three wave hindcast models in non-extreme conditions produced similar results with a low error against observed values. Campos and Soares (2016) concludes that the results supports the use of ERA-Interim for non-extreme analyses, HIPOCAS for high sea states in mid-high latitudes, close or within the storms, and NOAA/CFSR for surrounding areas and lower latitudes. Janssen (2002) performed a validation study using satellite data and found that present-day wave models, including WAM are reliable.

The actual observations at the locations would be preferable to hindcast data for fitting and training the models, but due to the lack of past observations and the enormous size of the ocean space, this is often not possible (Monbet et al., 2007). Observed data also has some uncertainties related to the measurement (Campos and Soares, 2016).

Thus, the foundation for building the stochastic generators rests on the inherent uncertainties and assumptions present in the hindcast models based on the WAM model. Assuming that the hindcast data is a sufficient representation of the wave conditions, then the uncertainty increases further when introducing models which provide additional abstraction of the wave process. An interesting extension of this thesis would be to compare the results of stochastic generators fitted or trained on hindcast data to actual observed data.

The WAM model is a physical model as opposed to the statistical metocean generators. ECWMF uses WAM in both deterministic and ensemble forecasting systems (Janssen, 2002). Vanem (2011) argues that waves should be modeled as stochastic processes since even though dynamics of waves follows the laws of physics and could in principle be described deterministically, this is not possible due to the complexity of the system. Vanem (2011) explains that the sea is a dynamic system which is influenced by an infinite number of interrelated parameters which needs to be determined to provide an exact representation of the sea.

Regardless, a metocean generator can not be fully deterministic since then it would generate one and only one synthetic time series and the evolution of the wave conditions would be similar given the same previous conditions. There must be some stochastic terms or other method which allows for the generation of several unique synthetic series.

7.1.2 Distribution of Observed T_p Values

North Sea Series Distribution

The observed North Sea T_p values were originally logarithmic spaced and then distributed within its class by the uniform distribution given by Equation 5.1 and 5.2. The joint distribution of the randomized observed data can be seen in Figure 6.11a. The classes can be recognized by characteristic vertical lines where two classes with a substantial difference in the number of occurrences are adjacent. This implies that the assumption of uniform distribution was not fully adequate.

The spreading can and should be improved such that the joint distribution no longer shows clear vertical lines between the classes. A simple way of doing this would be to introduce weights to a uniform distribution similar to the Mediterranean Sea distribution. The weight should be increased towards the neighboring class with the highest density. Regardless, the distribution method was found to be adequate for the models in this thesis.

Mediterranean Sea Series Distribution

The observed Mediterranean Sea T_p values were originally inconsistently logarithmic spaced and then distributed by merging some classes before applying a weighted uniform distribution function, as detailed in Section 5.1.2. Figure 6.11b shows the joint distribution, where vertical class lines starts to become visible for values of T_p higher than 8 seconds. The density of the markers before this point hides some of the vertical lines, but they are still present.

This could be improved by introducing a more comprehensive and robust method for both clustering and weighing the distribution. The distribution was found to be adequate for use in this thesis.

From the scatter plots in Figure 6.11, it can also be seen that the Mediterranean Sea series was less effected by swell than the North Sea series. There were few instances of high T_p for low

values of H_s for the Mediterranean Sea series compared to the North Sea series. This was reasonable as the Mediterranean Sea is much more enclosed than the North Sea. This is further corroborated by the discussion on curve fitting in Section 7.2.5.

7.2 Transformations and Residuals

The lognormal Rosenblatt transformation introduced in this thesis has so far not been applied in a traditional autoregressive simulation method. The general approach in the literature consists of using the Box-Cox transformation or the log transform (equivalent to the special case of Box-Cox where $\lambda = 0$) to produce approximately Gaussian residuals (Athanassoulis and Stefanakos, 1995; Stefanakos and Belibassakis, 2005; Soares and Cunha, 2000). The following section discusses the assumptions made for the transformations and how these assumptions influenced the models. The final subsection discusses the benefits and drawbacks of using the lognormal Rosenblatt transformation over the Box-Cox transformation.

7.2.1 Lognormal Assumption

A primary assumption for the lognormal Rosenblatt transformation was that both H_s and T_p follows a lognormal distribution. Det Norske Veritas (2010) states that T_p follows a lognormal distribution, while H_s should be modeled by a 3-parameter Weibull distribution. Weibull distribution was to be avoided since the resulting residuals follows the same distribution. The residuals must be Gaussian due to the assumptions underlying the VARMA models (Wei, 1990). In contrast to Det Norske Veritas (2010), Medina et al. (1991) states that both H_s and T_p can be modeled by a lognormal distribution.

The latter was followed in this thesis and produced excellent results for the VAR and VARMA models, with a significant improvement in the joint distribution, see Section 7.2.6 for further discussion.

7.2.2 Stationarity of Residuals

The stationarity of the residuals $\boldsymbol{W}$ can be checked by assessing a myriad of different test statistics. The stationarity of the residuals was assessed by the augmented Dickey–Fuller (ADF) (Fuller, 1976) and Kwiatkowski–Phillips–Schmidt–Shin (KPSS) (Kwiatkowski et al., 1992) test statistics.

ADF tests the null hypothesis that there is a unit root in the time series (Fuller, 1976). All transformed series rejects the null hypothesis of a unit root at the 5 % significance level. KPSS tests the null hypothesis that the time series is stationary around a deterministic trend (i.e. trend-stationary) against the alternative of a unit root (Kwiatkowski et al., 1992). All transformed series rejects the trend-stationary null hypothesis in favor of the unit root alternative at the 5 % significance level.

The two test statistics contradicts each other, but this is not uncommon for these two test statistics. These results simply shows that there is not sufficient information for determining if the

series are stationary (Bhargava, 1986). More complex test statistics could have been applied for further investigation, but this is outside the scope of this thesis. The results of the best performing models showed that the series were indeed stationary enough for practical applications, which is what truly matters.

7.2.3 Covariance of Residuals

An interesting result of the lognormal Rosenblatt transformation was that the monthly covariance between u_1 and u_2 became small and in some cases negative. Negative covariance complicates the seasonal transform due to the introduction of complex values from the inclusion of the square root of the covariance matrix.

Several tests were performed to find a solution, such as setting the diagonal to zero or taking the absolute value of the entries in the covariance matrix before applying the square root. It was found that giving the covariances a value of zero produced the best results. More work is needed to understand exactly why the covariance almost disappears after transformation. It is likely that the conditioning of the parameters is the cause of the low covariance.

7.2.4 Distribution of Residuals

Figure 6.12a depicts a scatter plot of the U-space residuals for the Mediterranean Sea series. The Mediterranean Sea scatter plot shows a cluster which has center close to the origin. There are only a few values far from the cluster, primarily located at higher u_2 values. Most of the values for both u_1 and u_2 have an absolute value of less than 4, which is reasonable. A distance of 4.5 from the origin is equivalent to the probability of a 10,000-year H_s or T_p.

Figure 6.12b depicts a scatter plot of the U-space residuals for the North Sea. The North Sea scatter plot shows clear horizontal lines of constant u_1 values. This is a result of the low fidelity of H_s values in the North Sea series. Further, it was seen that the change in H_s was small for increasing values of u_1 at low values, i.e., negative values. Much larger increases were seen in H_s for increasing values of u_1 at higher values. Additionally, the distribution of T_p does not influence the formation of the lines which corroborate that the low fidelity in H_s is the cause. The values of H_s could have been distributed before transformation, similar to T_p, but was not necessary for this thesis.

Most of the values of u_1 and u_2 have a distance of less than 4 from the origin, which again is reasonable. There are on the other hand, some values which even surpass the 10,000-year probability for T_p.

Figure 6.13 depicts the residual u_1 histogram for both observed series. The Mediterranean u_1 histogram shows a decent Gaussian distribution with some skewness. The North Sea u_1 residuals are centered around zero, but shows some "spikes" and empty bins. This is again a result of the low fidelity of H_s in the North Sea series. Distribution of H_s before transformation may improve the results.

Figure 6.14 depicts the residual u_2 histogram for both observed series. The histogram of Mediterranean u_2 again shows a decent Gaussian distribution with some skewness. The North Sea histogram of u_2 is skewed towards the right with a much fatter left than right tail.

The transformation produce the best results for the Mediterranean Sea series, where part of the increase in quality is a result of the higher fidelity of H_s. The variance is also lower for the Mediterranean Sea series which may also contribute to the results. The increase in quality of transformation does not directly relate to a better quality of results as discussed in Section 7.3.

7.2.5 Curve Fit

Figure 6.15 and 6.16 shows the curve fitting used for determining the parameters of the conditional mean and variance for the lognormal distribution of T_p. The curve fitting was found to be appropriate for both series. The largest deviations appear at the highest values of H_s for both series, but the generalization through these points appears appropriate. It should also be noted that there are quite few extreme values of H_s in each series which means that an exact fit through these points may not be the best generalization.

The figures further support the assumption that the North Sea series is more effected by swells. Comparing Figure 6.16a and 6.16b it is clear that the variance in $ln(T_p)$ for low values of H_s is much higher for the North Sea series than for the Mediterranean Sea series. The variance become more similar with increasing H_s.

This will effect further assumptions related to short-term modeling of waves in the two areas, especially for choice of wave spectra H_s and T_p can be used to describe. The swells significantly influence the shape of the wave spectra, i.e., one-peaked or two-peaked. For more information on the influence of swells on the choice of wave spectra see Haver and Moan (1983).

7.2.6 Rosenblatt Transformation Compared to Box-Cox Transformation

Appendix B.3 contains a comparison between two VARMA models with full seasonal transform on the North Sea series, where the comparison models uses different transforms to normality. VARMA(2,3)$_f$ has the lognormal Rosenblatt transformation as detailed in this thesis, while VARMA(3,1)$_{f,box}$ has the multivariate Box-Cox transformation (Box and Cox, 1964). The number of lags are different due to that the best candidate model changes with the transformation.

It can be seen that the Box-Cox model performs slightly better at replicating the overall lower order statistical moments. Both models perform equally well in matching the monthly mean, covariance and variance in H_s, but the Box-Cox model matches the monthly T_p variance much better. The figures show that the Rosenblatt model matches the crosscorrelation function slightly better, while the Box-Cox model more accurately matches the autocorrelation function of T_p. Both models perform equally well on matching the persistence above thresholds and H_s autocorrelation. The Box-Cox model perform slightly worse on matching the persistence below.

A noticeable difference occurs with the joint distribution. The Box-Cox model does not match the general shape of the observed distribution. The Box-Cox model generates values above the wave breaking limit and further does not produce low enough values of H_s for high values of T_p. The inward curve which appears for higher values of T_p is also not adequately matched by the Box-Cox model. The Rosenblatt model performs much better in matching the shape of joint distribution. Further, the Rosenblatt model also matches the marginal distributions slightly better.

In summary, the Rosenblatt transformation significantly increases the performance in matching the joint distribution, but performs slightly worse than Box-Cox on a few other criteria on the North Sea series. The literature shows that the Box-Cox transformation is generally applied for all applications (Monbet et al., 2007), but this thesis has shown that the lognormal Rosenblatt transformation should rather be used for applications where matching the joint sea state of H_s and T_p is of high importance. More work is needed to see if the transformation could be used for other metocean parameters.

7.3 Quality of Results

7.3.1 Overall Lower Order Statistical Moments

The overall mean is the primary criterion for many simulation-based design applications. For instance, when determining the operational life cycle cost of a deep-sea shipping vessel, the mean sea state that the vessel experiences is the primary characteristic for determining the fuel consumption.

Table B.1 and B.2 depicts the overall lower order statistical moments for the North Sea and Mediterranean Sea series, respectively. It can be seen that the Markov models most accurately matches the overall variances and covariance among all the models, but generates overall mean values which are below the observed value for both series.

The VAR and VARMA models most accurately match the overall mean values for both series. Note that VARMA(2,3)$_f$ generally performs best for the North Sea series, while VAR(30)$_f$ performs best for the Mediterranean Sea series. This indicates that it is beneficial to include the full seasonal transform when concerned with accurately replicating the overall lower statistical moments. There are no clear indications that VARMA outperform VAR on this criterion.

The SAR-LSTM model quite accurately matches the overall mean values for the North Sea, where the performance is comparable to that of the VAR and VARMA models. The SAR-LSTM model produces a much higher than observed variance and covariance, far higher than any other model. The SAR-LSTM network for the Mediterranean Sea series produce overall values significantly higher than observed and any other model. The matching is much worse for the variance and covariance compared to the mean for the SAR-LSTM networks. This is symptomatic of the general problem with the SAR-LSTM models, primarily that the introduction of the white noise to the LSTM network causes the variance to skyrocket. This will be further discussed in Section 7.6.2.

7.3.2 Monthly Lower Order Statistical Moments

The monthly mean and variance are good indicators for the ability of the model to recreate the seasonality in a times series. The winter season, compared to the summer season, normally results in harsher sea states and shorter persistence. Seasonality is highly important in several marine applications, especially for applications concerned with operability, such as marine construction, salvage, aquaculture and offshore wind power.

Monthly Mean

Figure B.1 and B.8 depicts the monthly mean for the North Sea and Mediterranean Sea series, respectively. The VAR and VARMA models accurately matches the monthly mean of H_s and T_p. There is a minor deviation near month 8 in the Mediterranean Sea series which is not present in the North Sea series. Further, the results are marginally better for the full transform models compared to the mean transform.

The Markov models accurately matches the monthly mean of H_s, but there are more significant deviations in the monthly values of T_p compared to VAR and VARMA.

The SAR-LSTM network for the North Sea series performs quite well in matching the monthly mean of both H_s and T_p. The performance is better than the Markov models and comparable to the VAR and VARMA models. This is not the case for the Mediterranean Sea, where the values are much larger for each month. The inverse bell shape resulting from the calmer period during summer is more pronounced for the Mediterranean Sea SAR-LSTM model.

Monthly Variance and Covariance

Figure B.2 and B.9 depicts the monthly variance and covariance for the North Sea and Mediterranean Sea series, respectively. The monthly variance and covariance is best matched by the VAR and VARMA models. For the North Sea series, it seems like the complex shape of the T_p variance makes it more difficult to match this compared to the Mediterranean Sea. None of the North Sea VAR and VARMA models quite manage to replicate this shape. The full seasonal transform results in higher monthly T_p variance than the mean transform for the North Sea series.

For the Mediterranean Sea series, the VAR and VARMA models quite accurately match the variance and covariance, except for some deviations in H_s variance and covariance during months 10-12. The full transform produces better results during these months compared to the mean transform. The full transform further only produce marginally better results than the mean transform, which is highly interesting, as a larger deviation between the models using different transforms was expected on this criterion.

The Markov models gave some interesting results on this criterion. The monthly T_p variance acts opposite of the observed behavior for both series. This is a result of the coupling matrix not having any tools for maintaining the seasonal variance in T_p and that only the mean transformation was used with the Markov models. The subpar performance on this criterion is therefore reasonable, since there are no tools for maintaining the seasonal variance. Further, the variance

in H_s and covariance are close to constant for both series. This was expected since the monthly variance in H_s and covariance should be the same as the overall value.

The SAR-LSTM models performs much worse than the other models. The SAR-LSTM model for the Mediterranean Sea produces monthly variances which are much higher than observed. The shape is maintained, but it is much steeper than observed. The North Sea series are better and maintains the shape, but the values are still far too high. This is particularly interesting since the SAR-LSTM network models use the mean transform. This indicates that the networks were able to learn the general shape of the monthly variance and covariance, but the values are excessive.

7.3.3 Joint and Marginal Distributions

Figure B.3 and B.10 depicts the joint and marginal distributions for the North Sea and Mediterranean Sea series, respectively. The matching of the marginal distribution means that the model is able to accurately recreate the number of times a significant wave height or peak period occurs over the temporal horizon under consideration. This includes the number of occurrences of extreme events which is of high importance for failure analysis or other extreme value analyzes.

Joint environmental models are required for a consistent treatment of the loading in for instance a reliability or operability analysis (Det Norske Veritas, 2010). A vessel may be limited by restrictions by each of the wind and sea state parameters or a combined state. Another major use of joint environmental models is in the assessment of the relative importance of various environmental variables during extreme response conditions (Det Norske Veritas, 2010).

The matching of the joint and marginal distributions is poor for the SAR-LSTM network models for both series. The matching is far below any of the other models. Higher values of H_s and T_p are much more prevalent and the models produce far more extreme values. The extreme values appear as elongated "tails" of the joint distributions. This hold for both series, where the North Sea series has two tails and the Mediterranean Sea series has only one. The extreme tails are most likely due to that the lognormal Rosenblatt transformation maps extreme values from simulation to the end of the tails, since the joint distribution shape is preserved for values close to observed values. The number of extreme values resulting from the simulation is significant such that the tails become very large.

The Markov models perform poorly in replicating the marginal distributions for both series. Some of the marginal distributions are jagged which is a consequence of the limited number of states. The joint distribution for both series are decent, but there are some distinct clusters which does not appear in the observed values

The VAR and VARMA models best replicates the joint and marginal distributions. The models with the mean seasonal transformation are slightly better at matching the marginal distributions, especially for T_p. The joint distribution is adequately matched by all the VAR and VARMA models, but the North Sea models are lacking some values near the lower right tail. There are further more extreme values in the joint distribution for both series. This is due to that the Gaus-

sian distributions can produce large values which when added to an already large value for the deterministic part of the model results in extreme values larger than observed.

7.3.4 Auto- and Crosscorrelation

Figure B.4 and B.11 shows the autocorrelation functions for the North Sea and Mediterranean Sea series, respectively. The autocorrelation and crosscorrelation functions gives a good indication of the ability of the model to replicate the dependence within a sequences of waves.

The autocorrelation function is sufficiently matched by the VAR and VARMA models for both series, and the deviations are smallest in the Mediterranean Sea series. There are no clear indication that the type of seasonal transformation influences the autocorrelation. Note that the autocorrelation function of the Mediterranean Sea drops off faster than the North Sea. This is reasonable since the time steps in the Mediterranean Sea series is 6 hours, twice that of the North Sea.

The autocorrelation function of the SAR-LSTM network models drops of much faster than the observed series and remains lower. The SAR-LSTM network models does not manage to replicate the temporal dependencies.

Both T_p autocorrelation functions for the Markov models have a sharp drop at the first lag. This effect is most pronounced for the North Sea series and is most likely due to the coupling matrix, since it samples based only on the current H_s. The autocorrelation functions for H_s have a higher autocorrelation than the observed autocorrelation functions for lower lags. This may be due to the limited number of possible states which can be entered, thus increasing the past autocorrelation even when the process is Markovian.

Figure B.5 and B.12 shows the crosscorrelation functions for the North Sea and Mediterranean Sea series, respectively. The crosscorrelation function in the Markov models is mirrored around lag 0 and thereby shows an equal dependence on past and present lags. The crosscorrelation in the LSTM networks is also centered at lag 0, but is not mirrored in the same way as the Markov models. Both crosscorrelation function for the SAR-LSTM network models drops off much faster than any other model and observed function.

The VAR and VARMA models perform much better in matching the crosscorrelation than the Markov models and SAR-LSTM networks. The fit for the smallest lags, [-5,5], appears to improve with the full seasonal transformation for the Mediterranean Sea series, while the fit worsens for the North Sea series. The VAR models performs best which may indicate that the moving-average (MA) component does not improve the crosscorrelation.

7.3.5 Persistence of Significant Wave Height

Persistence of sea states is an important aspect in the simulation of many types of marine operations. This is especially true for applications where some operation has to be performed within a time window restricted by some environmental parameter(s). When concerned with this type

of application it is common to use the terminology access window and waiting window for persistence below threshold and persistence above threshold, respectively. This terminology will be used for the remainder of the thesis to connect the persistence to one of its intended end-use.

Waiting Windows

Figure B.6 and B.13 depicts the normalized contours for persistence above threshold for the North Sea and Mediterranean Sea series, respectively. The SAR-LSTM network models produce too many waiting windows for all values of H_s for the Mediterranean Sea and far too few for the North Sea series. The matching is best for the low contours of the Mediterranean Sea series. The contours for the North Sea series are more horizontal than the observed, which shows that too few long waiting windows are produced for low threshold values.

The Markov models acts opposite of the SAR-LSTM network models for the Mediterranean Sea series and produces too many waiting windows for all values of H_s. For the North Sea series, the Markov models also produce too many waiting windows for low threshold values. The contours for both series are further much steeper, meaning that far too many long waiting windows are produced for lower values of H_s. This was expected due to the lack of memory in the Markov models.

The VAR and VARMA models accurately matches the waiting window contours, far superior to the other models. The matching increases with increasing contour levels and the fit is exceptional for contour levels higher than 0.01. This is to be expected since there are few occurrences of waiting windows at the lowest contours levels and matching these extreme values is much more difficult. The North Sea models produce too many long waiting windows for the higher threshold values, while the Mediterranean Sea models produce too many long waiting widows for the highest thresholds and too few long waiting windows for threshold values between 2-3 meters.

Access Windows

Figure B.6 and B.13 shows the normalized contours for persistence below threshold for the North Sea and Mediterranean Sea series, respectively. The SAR-LSTM network models performs even worse on matching the access windows than waiting windows. This is most pronounced in the Mediterranean series where the model produces far less access windows for all threshold values. The North Sea model produce too many short access windows for thresholds less than 1.5-2 meters and and far less long access windows for threshold larger than 2 meters.

The Markov models performed again better than the SAR-LSTM network models. The Markov models for both series produces too many access windows for all threshold values. They were also much steeper, meaning that they produce too many long access windows for higher values of H_s.

The matching of the access windows is excellent for the North Sea models. The largest deviation occurs at the lowest contour levels as expected. The performance of the Mediterranean

Sea model is not as good as the North Sea series and worse than the performance seen for the waiting windows. The models produces the same amount of access windows for lowest thresholds, but produces more and longer access windows for thresholds larger than approximately 1.5 meters.

The VAR and VARMA models performed very well and quite equally on both waiting and access windows, which indicates that the persistence performance is not effected by the type of seasonal transform. The Mediterranean Sea series has lower maximum contours for waiting windows compared to the North Sea, which is reasonable due to the calmer climate. Further, this discussion shows that using this simple persistence criteria was adequate for evaluating the persistence performance of the models.

The increased persistence performance on the North Sea series may be a consequence of the shorter sampling frequency. The Mediterranean Sea series has a sampling frequency of 6 hours, which is at the top of range for recommended sampling frequencies according to Det Norske Veritas (2010). The sampling frequency for the North Sea series is 3 hours, which is the industry standard sampling frequency (Det Norske Veritas, 2010).

7.4 Practicality

The Markov models are entirely described by only a relatively small transition matrix and some method for maintaining the dependencies between variables. The time and resources required for implementing a Markov model is negligible. This makes the Markov models practical for use in a simulation-based design context, since it is generally necessary to generate several models for a single simulation. It is further easy to find the optimal order of the model, since this is generally the maximum number of states before absorbing states appear.

VARMA models requires more time for implementation than the Markov models. This is due to that it is computationally expensive to estimate the model parameters by maximum likelihood estimation. The computational cost increases with the length of the series used for fitting. The computational cost sets a limit for the maximum number of p, q lags and the number of candidate models to evaluate. The number of parameters to be estimated increases rapidly with increases in number of p or q lags. The literature shows that generally low order VARMA models are evaluated due to the computational cost (Stefanakos and Belibassakis, 2005) (Monbet et al., 2007).

VAR models does not exhibit the same rapid increase in computational cost for increases in number of p lags, due to the lower number of total parameters. Thus, VAR models are much more practical in this context than VARMA models. Further, the low cost of estimation for VAR models makes it possible to estimate all candidate models and then use a selection method such as AIC or BIC. These selection methods are generally considered to be better estimators of the relative quality of the model compared to using the P-values of the extended crosscorrelation matrix (Burnham and Anderson, 2002).

The time required for VAR and VARMA estimation depends on the hardware and algorithm used. For instance, it took the author 1-3 minutes to estimate the parameters of a VAR model compared to 4-6 hours for a VARMA model. It should be noted that these computations were done on a laptop with only 8 GB RAM and an i5-7200U processor. Increases in computational power will bring the estimation time of VARMA down, but there is still a sizable difference in the computational cost.

SAR-LSTM network models requires significant time and computational resources. The sheer number of possible hyperparameters and settings for the hyperparameters of a deep neural network model makes it a daunting task. In order to determine the values of the hyperparameters it is necessary to carry out some sort of search procedure, normally random or grid based (Goodfellow et al., 2016). This is highly computationally expensive and may require several iterations to find a "good enough" solution. Performing several grid searches for all relevant spatial locations for a single simulation run may be too computationally expensive to be practical. This is of course dependent on the availability of computational power. Further, the effects of different metocean characteristics on the deep learning algorithms are not yet understood.

It is the author's opinion that for deep neural networks to be practical in a simulation-based design context there must be a joint effort to create a shared library containing the model specifications and parameters for time series at specific locations. A library of models would be required for performing high spatial fidelity simulation in a timely manner. This is a consequence of the time required for training a single model and would further require a shared standard for validation and algorithm design which may be troublesome.

Storing the parameters of any model family is not considered to be an issue. SAR-LSTM network models have far more parameters than any of the other families, but it is still not a significant number of values. This holds true even when it is required to store the parameters of several models.

The difference in time and resources required for the simulation of a synthetic series is not significantly different for the models. SAR-LSTM requires a bit more computation than VAR and VARMA, which again is slightly more computationally demanding than Markov models. Regardless, the difference is negligible for use in simulation-based design.

7.5 Ranking of Models

The VAR and VARMA models were far superior to the other model families in the quality of results and the difference in the quality of results were minor for both model types. VAR models performed better than VARMA models on some criteria, while VARMA models were better than VAR models on others. The added complexity from including the MA component did not result in a significant difference in the quality of results. A VARMA model did perform better overall on the North Sea series, while a VAR model did best on the Mediterranean Sea series. This shows that the choice between VARMA and VAR when concerned with the quality of results is dependent on the observed series.

The type of transformation applied also depends on which criteria is of the highest interest for the end-user. The full seasonal transformation was shown to have the best overall results, but the mean transform did perform marginally better on some criteria. This means that the choice of model type and transformation is depended on the intended end-use of the synthetic series and the observed time series in question. Further, VAR models are significantly more practical to use than VARMA models due to the lower computational cost for parameter estimation. If the end-user requires a significant amount of different stochastic generators then VAR may be a better choice of model due to its higher practicality.

The Markov models produced results of a quality higher than the SAR-LSTM networks, but below that of the VAR and VARMA models. Markov models requires the least amount of effort to implement, but the trade-off in quality of results make them subpar to the VAR and VARMA models. It should be noted that the Markov models introduced in the preceding project thesis performed better than the Markov model presented here, but the results are still subpar to the VAR and VARMA models.

The SAR-LSTM networks performed worst in both quality of results and practicality. The practicality of the models is low due to the investment required for training a single stochastic generator. See Section 7.6 below for further discussion on deep learning in simulation-based design.

7.6 Deep Learning in Simulation-based Design

7.6.1 Training and Validation Error

The validation and training error of the deep learning models is depicted in Figure 6.10, and followed the expected behavior as seen in Goodfellow et al. (2016). There were no indication of overfitting or underfitting, which was corroborated by that the grid search showed that the best models did not require any L^2 norm penalty, see Table 6.1. Further, from Figure 6.10 it can be seen that the validation error falls below that of the training error, which is due to the dropout regularization being disabled during validation.

7.6.2 Stochastic Autoregressive LSTM Network Models

The stochastic autoregressive LSTM network model introduced in this thesis did not perform satisfactory, but it is important to note that other deep neural networks may produce results of the desired quality. Aminzadeh-Gohari et al. (2008) showed that it was indeed possible to simulate the univariate series of H_s using neural networks. However, the practicality of such a network in simulation-based design was not investigated in this thesis. The approach of Aminzadeh-Gohari et al. (2008) differs significantly from the methods introduced in this thesis and it may be the case that the autoregressive white noise method is unfeasible for deep learning. More work on this subject is needed to understand why the networks produce results of such low quality.

It is important to note that the LSTM network models performed excellent in prediction. Figure 6.9 shows the prediction results for a part of the unseen North Sea test set. The model predicted H_s with a low error, but was a bit weaker in predicting T_p. Table 6.4 contains the RMSE of each of the variables and shows that the RMSE was more than three times higher for T_p. This is most likely a result of the higher variance in T_p compared H_s. The variance in T_p is almost three times that of H_s. This can also be seen in Figure 6.9 where T_p has far more erratic behavior. The higher variance makes it harder for the deep learning network to learn the representation.

The networks predict adequately, but their simulation performance is far below the required quality. The main problem is that the variance of the outputs increases far beyond that of the observed values, see Table B.1 and B.2. Directly adding Gaussian white noise to the outputs, and then feeding back to the inputs, did not perform as expected. More work is needed to understand exactly why the white noise leads to such subpar results.

The complexity of deep learning makes it difficult to even write a short list of the different options that could be further explored. Different model types, optimization algorithms, hyperparameters, transformations and so on could change the performance of the deep learning algorithm.

The amount of possible options leads to a low practicality for simulation-based design, unless a shared "standard" configuration is found, which produces satisfactory results for different spatial and temporal scales. The amount of resources required for testing a single network configuration limits further sets limitations for the practicality in simulation-based design.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

The objective of the master's thesis was to validate the quality of deep learning models as stochastic metocean generators and their practicality in a simulation-based design context. To answer the research questions, a new type of deep learning model called stochastic autoregressive long short-term memory (SAR-LSTM) network model was developed. The SAR-LSTM model was validated in a case study against traditional stochastic metocean generators; Markov chain, VAR and VARMA models. The case study consisted of simulating and matching selected statistical criteria of two bivariate series of H_s and T_p from different spatial locations and hindcast archives.

Validation of the quality of results showed that the SAR-LSTM network models did not adequately match the statistical criteria. The SAR-LSTM networks produced an excessive amount of variance in the synthetic series which resulted in the subpar results. The VAR and VARMA models performed adequately in matching the statistical criteria and were found to give a sufficient description of H_s and T_p. The Markov chain models performed better than the SAR-LSTM model, but were subpar to the VAR and VARMA models and found to not give a sufficient description of the wave conditions.

Thus, the conclusion to the first research question is that the stochastic autoregressive LSTM network models are not adequate as stochastic metocean generators and produces results of a much lower quality than simpler traditional stochastic metocean generators. Note that this only applies the SAR-LSTM model not all deep learning stochastic metocean generators.

The practicality of the models in a simulation-based design context was then validated based on the work done in the case study. The SAR-LSTM models were found to be unpractical due to the significant time and computational cost for implementing a single stochastic generator. The Markov models had the highest practicality due to only being described by an easy to compute transition matrix and coupling matrix. The VAR and VARMA models had a high practicality, but the practicality of the VAR models was more favorable than the VARMA models due to the lower computational cost for estimating the model parameters. Storage of the model parameters was found to be practical for all model types.

Thus, the conclusion to the second research question is that deep learning stochastic metocean generators are not practical in simulation-based design, unless only a very limited set of unique metocean generators are required.

The VAR and VARMA models were found to be the best performing models since they had the best quality of results and were highly practical. The VARMA models showed some marginal improvements over the VAR models on some criteria for both series and the extension can be used if computation time is not constrained.

The type of seasonal transformation was found to be dependent on the intended application of the synthetic series. Finally, it can be concluded that the lognormal transformation introduced in this paper should be used over the Box-Cox transformation in metocean simulation if matching the joint distribution of H_s and T_p is of importance in the application.

8.2 Further Work

The excellent prediction performance of the SAR-LSTM network showed that the deep learning stochastic metocean generator was indeed able to learn the representation for predicting H_s and T_p. The poor simulation results must then be the result of the Gaussian white noise approach. Advice and assistance was solicited from PhD candidates and professors at the Department of Computer Science (IDI) of NTNU, but they were unable to provide any answers to the variance problem in the allotted time. Further work should be initiated into investigating why adding Gaussian noise to the predictions results in such extreme variance.

It may also be that another type of deep learning model is better suited for use as the prediction component in such an autoregressive simulation method. This should also be further investigated.

The lack of computational power led to restrictions for the grid search. Different hyperparameters, other optimization methods, larger network, training on noisy series, scaling and other changes to the algorithm may lead to a deep network which performs adequately. There is a tremendous amount of options that can be further investigated, but this is again a reflection of the low practicality of deep learning in this context. Investigating all possible options is not feasible. The grid search results showed that more hidden states and longer window length should be investigated.

The lognormal Rosenblatt transformation introduced in this thesis showed great improvements in describing the joint distribution. Further work should be initiated into validating this type of transformation for other metocean conditions and in higher multivariate cases for the VAR and VARMA models.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Ailliot, P. and Monbet, V. (2012). Markov-switching autoregressive models for wind time series. *Environmental Modelling & Software, Volume 30, Pages 92-101*.

Akaike, H. (1973). Information theory and an extension of the maximum likelihood principle. *2nd International Symposium on Information Theory, Tsahkadsor, Armenia, USSR, September 2-8, 1971, Budapest: Akadémiai Kiadó, pp. 267–281*.

Aminzadeh-Gohari, A., Bahai, H., and Bazargan, H. (2008). Simulation of significant wave height by neural networks and its application to extreme wave analysis. *Journal of Atmospheric and Oceanic Technology Volume 26*.

Anastasiou, K. and Tsekos, C. (1996). Persistence statistics of marine environmental parameters from markov theory, part1: Analysis in descrete time. *Applied Ocean Research, 18, 187-199*.

Andersen, O. J. (2009). The peak period in the wam10 hindcast archive. *Statoil*.

Athanassoulis, G. A. and Stefanakos, C. (1995). A nonstationary stochastic model for long-term time series of significant wave height. *Journal of Geophysical Research Atmospheres*.

Balluff, S., Bendfeld, J., and Krauter, S. (2015). Short term wind and energy prediction for offshore wind farms using neural networks. *4th Internationa Conference on Renewable Energy Research and Applications*.

Bazargan, H., Bahai, H., and Aryana, F. (2007). Simulation of the mean zero-up-crossing wave period using artificial neural networks trained with a simulated annealing algorithm. *Journal of Marine Science and Technology, March 2007, Volume 12, Issue 1, pp 22–33*.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE Trans. PAMI, special issue Learning Deep Architectures. 35: 1798–1828*.

Bettencourt, J. (1993). Characterisation and prediction of wave power resources. Master's thesis, FCUL, Lisbon Classical Univ.

Bhargava, A. (1986). On the theory of testing for unit roots in observed time series. *The Review of Economic Studies. 53 (3): 369–384.*

Borgman, L. E. and Scheftne, N. W. (1991). Simulation of time sequences of wave height, period, and direction. *Tech. Rep. DRP-91-2, 217 pp., U.S. Army Corps of Eng., Washington, D.C.*

Box, G. and Cox, D. (1964). An analysis of transformations. *Journal of the Royal Statistical Society, Series B. 26 (2): 211–252.*

Brockwell, P. and Davis, R. (2009). *Time Series: Theory and Methods.* Springer.

Brownlee, J. (2017). *Long Short-Term Memory Networks With Python.* E-book.

Burnham, K. P. and Anderson, D. R. (2002). *Model Selection and Multimodel Inference: A practical information-theoretic approach (2nd ed.).* Springer-Verlag.

Cadenas, E. and Rivera, W. (2009). Short term wind speed forecasting in la venta, oaxaca, méxico, using artificial neural networks. *Renewable Energy Volume 34, Issue 1, January 2009, Pages 274-278.*

Caires, S. and Sterl, A. (2005). A new non-parametric method to correct model data: Application to significant wave height from the era-40 reanalysis. *J. Atmospheric and Oceanic Tech., In press.*

Campos, R. and Soares, C. (2016). Comparison and assessment of three wave hindcasts in the north atlantic ocean. *Journal of Operational Oceanography Volume 9 - Issue 1.*

Camus, P., Méndez, F., and Medina, R. (2011). A hybrid efficient method to downscale wave climate to coastal areas. *Coastal Engineering 58, 851–862.*

Chollet, F. (2015). Keras. `https://keras.io`.

Det Norske Veritas (2010). *Recommended Practices DNV-RP-C205.* Det Norske Veritas.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12 2121-2159.*

ECMWF (2018). Era5. `https://www.ecmwf.int/en/forecasts/datasets/archive-datasets/reanalysis-datasets/era5`.

Erikstad, S. (2017). Merging physics, big data analytics and simulation for the next-generation digital twins. In *HIPER 2017, High-Performance Marine Vehicles.*

Fathi, D., Grimstad, A., Johnsen, T., Nowak, M., and Stålhane, M. (2013). Integrated decision support approach for ship design. *Oceans 2013 conference.*

Ferreira, J. and Guedes Soares, C. (1999). Estimation of the occurrence of severe sea states with beta distributions. *Ocean Eng. 26 8., 713–725.*

Fuller, W. (1976). *Introduction to Statistical Time Series*. John Wiley and Sons.

Gaspar, H. M. (2013). *Handling Aspects of Complexity in Conceptual Ship Design*. PhD thesis, NTNU.

Ghasemi, F., Mehridehnavi, A., Fassihi, A., and Perez-Sanchez, H. (2017). Deep neural network in biological activity prediction using deep belief network. *Applied Soft Computing. 62: 251*.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Graves, A., Abdelrahman, M., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778.

Guanche, Y., Mínguez, R., and Méndez, F. (2013). Climate-based monte carlo simulation of trivariate sea states. *Coastal Engineering 80, 107–121*.

Hagen, A. and Grimstad, A. (2010). The extension of system boundaries in ship design. *Transactions of RINA, vol. 152*.

Hagen, B., Simonsen, I., Hofmann, M., and Muskulus, M. (2013). A multivariate markov weather model for o&m simulation of offshore wind parks. *10th Deep Sea Offshore Wind R&D Conference, DeepWind 2013*.

Hamilton, J. (1989). A new approach to the economic analysis of nonstationary time series and the business cycle. *Econometrica, 57:357–384*.

Hasselmann, K., Barnett, T., Bouws, E., Carlson, H., Cartwright, D., Enke, K., Ewing, J., Gienapp, H., Hasselmann, D., Kruseman, P., Meerburg, A., Mller, P., Olbers, D., Richter, K., Sell, W., and Walden, H. (1973). Measurements of wind-wave growth and swell decay during the joint north sea wave project (jonswap)'. *Ergnzungsheft zur Deutschen Hydrographischen Zeitschrift Reihe, A(8) (Nr. 12), p.95*.

Haver, S. and Moan, T. (1983). On some uncertainties related to the short term stochastic modelling of ocean waves. *Applied Ocean Research Volume 5, Issue 2, April 1983, Pages 93-108*.

Hering, A., K., K., and Kleiber, W. (2015). A markov-switching vector autoregressive stochastic wind generator for multiple spatial and temporal scales. *Resources 2015, 4, 70-92; doi:10.3390/resources4010070*.

Hinton, G. (2012). Neural networks for machine learning. In *RMSprop*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation 9 (8): 1735-1780*.

Hogben, N. and Standing, R. (1987). A method for synthetising time history data from persistence statistics and its use in operational modelling. *Underwater Technology, Society for Underwater Technology, 13(4), 11-18*.

Huang, Z. and Chalabi, Z. (1995). Use of time series analysis to model and forecast wind speed. *J. Wind Eng. Ind. Aerodyn., 56, 311-322*.

Izquierdo, P. and Guedes, S. C. (2005). Analysis of sea waves and wind from x-band radar. *Ocean Engineering, 32(11-12), 1404-1419*.

Janssen, P. (2002). The wave model. `https://www.ecmwf.int/en/elibrary/16951-wave-model`.

Jim, J. Z. and Chou, C. R. (2002). A study on simulating the time series of significant wave height near the keelung harbour. *Proceedings of the 12th International Offshore and Polar Engineering Conference, (2002), 92-96*.

Keskar, N., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. (2016). Learning on large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Kirkwood, J. (2015). *Markov Processes*. Taylor & Francis Group, LLC.

Komen, G., Cavaleri, L., Donelan, M., Hasselman, K., Hasselman, S., and Janssen, P. (1996). *Dynamics and Modelling Ocean Waves*. Cambride University Press.

Konstant, D. G. and Piterbar, V. I. (1993). Extreme values of the cyclostationary gaussian random process. *J. Appl. Probab., 30, 82-97*.

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. *NIPS 2012: Neural Information Processing Systems*.

Kwiatkowski, D., Phillips, P., Schmidt, P., and Shin, Y. (1992). Testing the null hypothesis of stationarity against the alternative of a unit root. *Journal of Econometrics 54 (1992) 159-178. North-Holland*.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature volume 521, pages 436–444*.

Lehmann, E. L. and Casella, G. (1998). *Theory of Point Estimation (2nd ed.)*. Springer.

Leontaris, G., Morales-Nápoles, O., and Wolfert, A. (2016). Probabilistic scheduling of offshore operations using copula based environmental time series – an application for cable installation management for offshore wind farms. *Ocean Engineering 125 (2016) 328–341*.

Li, G. and Jing, S. (2010). On comparing three artificial neural networks for wind speed forecasting. *Applied Energy Volume 87, Issue 7, July 2010, Pages 2313-2320*.

Lopez, E., Valle, C., and Allende, H. (2016). Recurrent networks for wind speed forecasting. *International Conference on Pattern Recognition Systems (ICPRS-16), 2016 page 12 (6 .)*.

Makarynskyy, O., Pires-Silva, D., Makarynska, D., and Ventura-Soares, C. (2004). Artificial neural networks in wave prediction at the west coast of portugal. *Computer & Geosciences, In press*.

Martin, V., Hurn, S., and Harris, D. (2012). *Econometric Modelling with Time Series: Specification, Estimation and Testing*. Cambridge University Press.

Medina, J. R., Gimenez, M. H., and Hudspeth, R. T. (1991). A wave climate simulator, in proceedings of the 24th international association of hydraulic research congress, pp. b.521-528. *International Association of Hydraulic Research*.

Mitchell, T. (1997). *Machine Learning*. McGraw-Hill, Inc.

Monbet, V., Ailliota, P., and Prevostob, M. (2007). Survey of stochastic models for wind and sea state time series. *Probabilistic Engineering Mechanics Volume 22, Issue 2, April 2007, Pages 113-126*.

Monfared, M., Rastegara, H., and Kojabadi, H. (2009). A new strategy for wind speed forecasting using artificial intelligent methods. *Renewable Energy Volume 34, Issue 3, March 2009, Pages 845-848*.

Morales, J., Mínguez, R., and Conejo, A. (2010). A methodology to generate statistically dependent wind speed scenarios. *Applied Energy 87, 843–855*.

More, A. and Deo, M. (2003). Marine structures, 16 , 35-49. *Forecasting wind with neural networks*.

Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted boltzmann machines. *ICML'10 Proceedings of the 27th International Conference on International Conference on Machine Learning Pages 807-814*.

Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press.

Norwegian Meterological Institute (2009). A high-resolution hindcast of wind and waves for the north sea, the norwegian sea and the barents sea. *Norwegian Meterological Institute Report no. 2009/14*.

O'Carroll, F. (1984). Weather modelling for offshore operations. *The Statistician, vol. 33, pp. 161-169*.

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2013). How to construct deep recurrent neural networks. *CoRR*, abs/1312.6026.

Patterson, J. and Gibson, A. (2017). *Deep Learning: A Practitioner's Approach 1st Edition*. O'Reilly Media.

Percival, D. (1993). *Spectral Analysis for Physical Applications*. Cambridge University Press.

Pierson, W. J. J. and Moskowitz, L. (1964). A proposed spectral form for fully developed seas based on the similarity theory of s. a. kitaigorodskii. *Journal of Geophysical Research, Vol. 69, p.5181-5190*.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics, Vol. 22, No. 3., pp. 400-407*.

Rosenblatt, M. (1952). Remarks on a multivariate transformation. *Annals of Mathematical Statistics 23, 470–472*.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning representations by back-propagating errors. *Nature volume 323, pages 533–536*.

Schaul, T., Antonoglou, I., and Silver, D. (2013). Unit tests for stochastic optimization. *CoRR*, abs/1312.6055.

Scheu, M., D. Matha, D., Hofmann, M., and Muskulus, M. (2012a). Maintenance strategies for large offshore wind farms. *Deep Sea Offshore Wind R&D Conference 24 (2012), 281-288*.

Scheu, M., Matha, D., and Muskulus, M. (2012b). Validation of a markov-based weather model for simulation of o&m for offshore wind farms. *Proceedings of the 22th International Offshore and Polar Engineering Conference, (2012), 463-368*.

Scotto, M. and Guedes Soares, C. (2000). Modelling the long-term time series of significant wave height with non-linear threshold models. *Coastal Engineering Volume 40, Issue 4, July 2000, Pages 313-327*.

Soares, G. and Cunha, C. (2000). Bivariate autoregressive models for the time series of significant wave height and mean period. *Coastal Engineering 40 2000. 297–311*.

Spellman, F. (2016). *Science of Renewable Energy*. Taylor and Francis Group.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15 (2014) 1929-1958*.

Stefanakos, C. and Athanassoulis, G. (2003). Bivariate stochastic simulation based on nonstationary time series modelling. *13th International Offshore and Polar Engineering Conference, ISOPE'2003, Honolulu, Hawaii, USA, pp. 46-50*.

Stefanakos, C. and Belibassakis, K. (2005). Nonstationary stochastic modelling of multivariate long-term wind and wave data. *24th International Conference on Offshore Mechanics and Arctic Engineering (OMAE 2005)*.

Stefanakos, C. N. and Athanassoulis, G. A. (2001). A unified methodology for the analysis, completion and simulation of nonstationary time series with missing-values, with application to wave data. *Applied Ocean Research, vol. 23/4, pp. 207-220*.

Stephos, A. (2000). A comparison of various forecasting techniques applied to mean hourly wind time series. *Renewable Energy, 21, 23-35*.

Tiao, G. and Tsay, R. (1983). Multiple time series modeling and extended sample cross-correlations. *Journal of Business & Economic Statistics, Vol. 1, No. 1 (Jan., 1983), pp. 43-56*.

Toarmina, R., Chau, K., and Sethi, R. (2012). Artifical neural network simulation of hourly groundwater levels in a coastal aqufier system of the venice lagoon. *Engineering Applications of Artificial Intelligence 25 (2012) 1670–1676*.

Toll, R. (1997). Autoregressive conditional heteroscedasticity in daily wind speed measurements. *Theor. Appl. Climatol., 56, 113-122.*

Torsethaugen, K. (1996). Model for a doubly peaked wave spectrum. *SINTEF report STF22 A96204.*

Tsai, C., Lin, C., and Shen, J. (2002). Neural network for wave forecasting among multi-stations. *Ocean Engineering 29 (2002) 1683–1695.*

Tsay, R. (2014). *Multivariate Time Series Analysis with R and Financial Applications.* John Wiley. Hoboken, NJ.

Vanem, E. (2011). Long-term time-dependent stochastic modelling of extreme waves. *Stochastic Environmental Research and Risk Assessment, Volume 25, Issue 2, pp 185-209.*

Vanem, E. (2016). Copula-based bivariate modelling of significant wave height and wave period and the effects of climate change on the joint distribution. *Proceedings of the ASME 2016 35th International Conference on Ocean, Offshore and Arctic Engineering.*

Vik, I. (1981). The tile series generating module-modelling aspects ocean research- operational criteria. *CNRD 3-2 task 2.*

Walker, R., Nieuwkoop-McCall, J., Johanning, L., and Parkinson, R. (2013). Calculating weather windows: Application to transit, installation and the implications on deployment success. *Ocean Engineering 68 (2013) 88–101.*

Walton, T. and Borgman, L. (1990). Simulation of non-stationary, non-gaussian water levels on the great lakes. *J. of Waterways, Ports, Coastal and Ocean Division, ASCE, 116(6).*

Wei, W. (1990). *Time Series Analysis: Univariate and Multivariate Methods.* Pearson Education, Inc.

Wilson, D. and Martinez, T. (2013). The general inefficiency of batch training for gradient descent learning. *Nerual Networks 16(10):1429-51.*

Woolf, D. and Challenor, P. (2002). Statistical comparison of satellite and model waves climatologies. *Proc. 4th symp. WAVE.*

Xiaoyun, Q., Xiaoning, K., Chao, Z., Shuai, J., and Xiuda, M. (2016). Short-term prediction of wind power based on deep long short-term memory. *2016 IEEE PES Asia-Pacific Power and Energy Conference.*

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B: 301–320.*

Appendices

Appendix A

List of Acronyms

ADF Augmented Dickey–Fuller

AIC Akaike Information Criterion

ANN Artificial Neural Network

AR Autoregressive

ARMA Autoregressive Moving-average

BIC Bayesian Information Criterion

FNN Feedforward Neural Network

GARCH Generalized Autoregressive Conditional Heteroskedasticity

HMM Hidden Markov-Switching Models

KPSS Kwiatkowski–Phillips–Schmidt–Shin

LSTM Long Short-term Memory

MA Moving-average

MLE Maximum Likelihood Estimation

MLP Multilayer Perceptron

MS-AR Markov-Switching Autoregressive

MS-VAR Markov-Switching Vector Autoregressive

SETAR Self-Exciting Threshold Autoregressive

SVM Support Vector Machine

TAR Time-Varying Autoregressive

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

VARMA Vector Autoregressive Moving-average

Appendix B

Validation Results

B.1 North Sea Validation Results

Table B.1: Overall Lower Order Statistical Moments for North Sea Series

Series	Mean H_s	Variance H_s	Mean T_p	Variance T_p	Covariance
Observed	2.081	1.688	7.786	4.864	1.494
Markov$_m$	2.342	1.746	8.126	4.572	1.415
LSTM Network$_m$	2.050	7.723	7.984	13.321	5.724
VAR(7)$_m$	2.116	2.103	7.796	4.472	1.820
VAR(7)$_f$	2.134	2.233	7.857	5.292	1.912
VARMA(3,3)$_m$	2.111	2.025	7.789	4.359	1.723
VARMA(2,3)$_f$	2.106	2.047	7.840	5.003	1.708

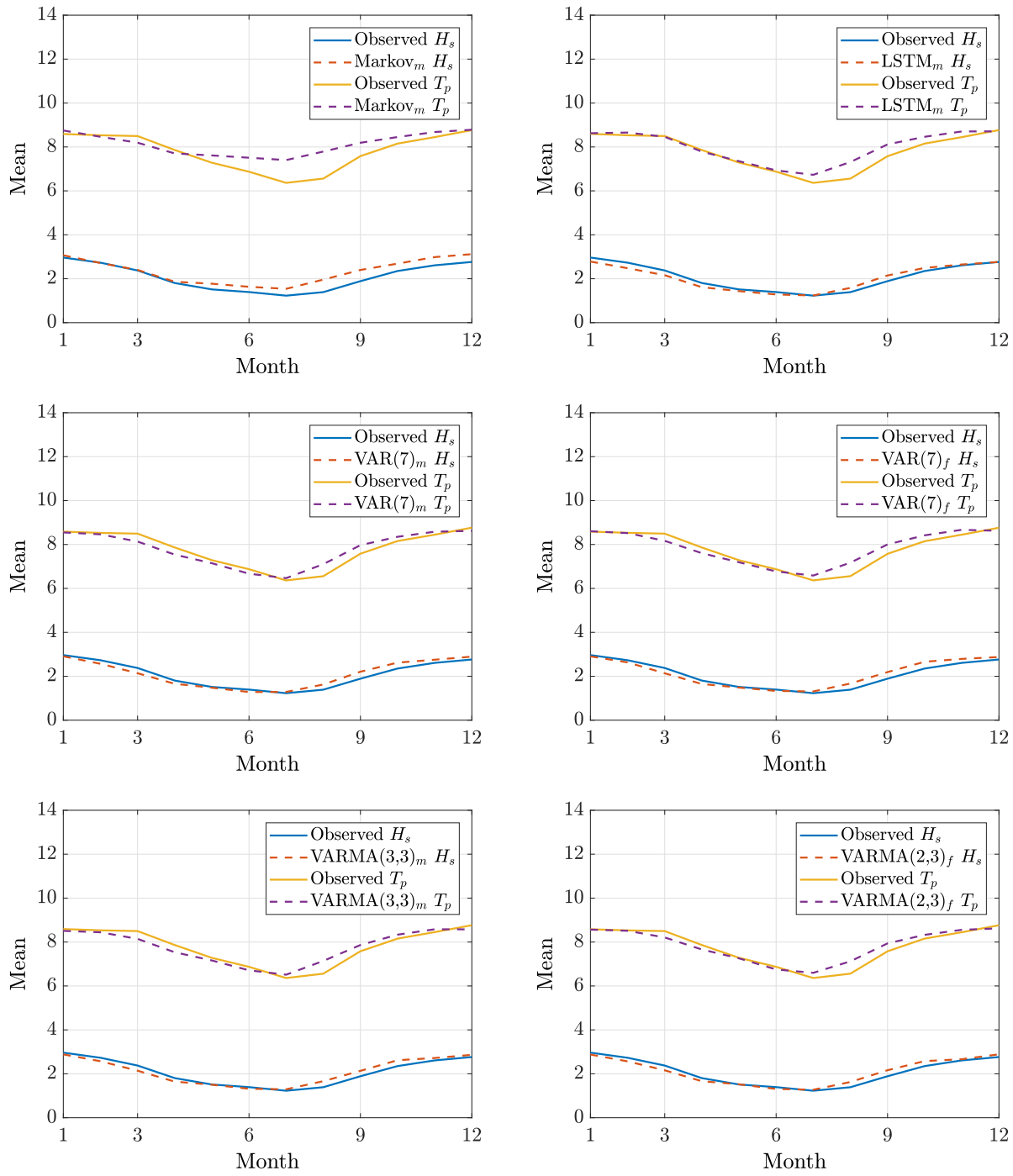


Figure B.1: Comparison Plot of Monthly Mean for North Sea Series

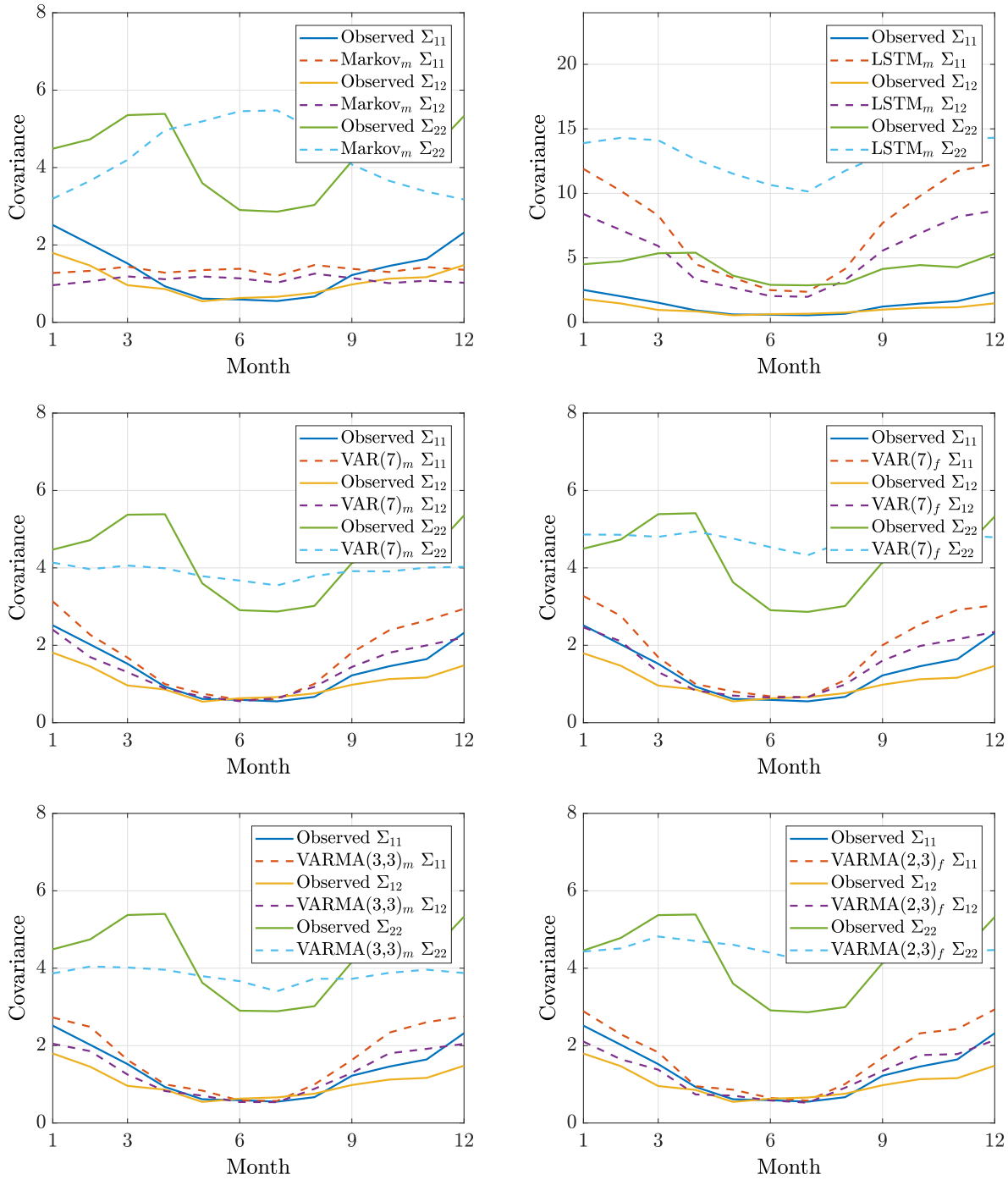


Figure B.2: Comparison Plot of Monthly Variances and Covariance for North Sea Series

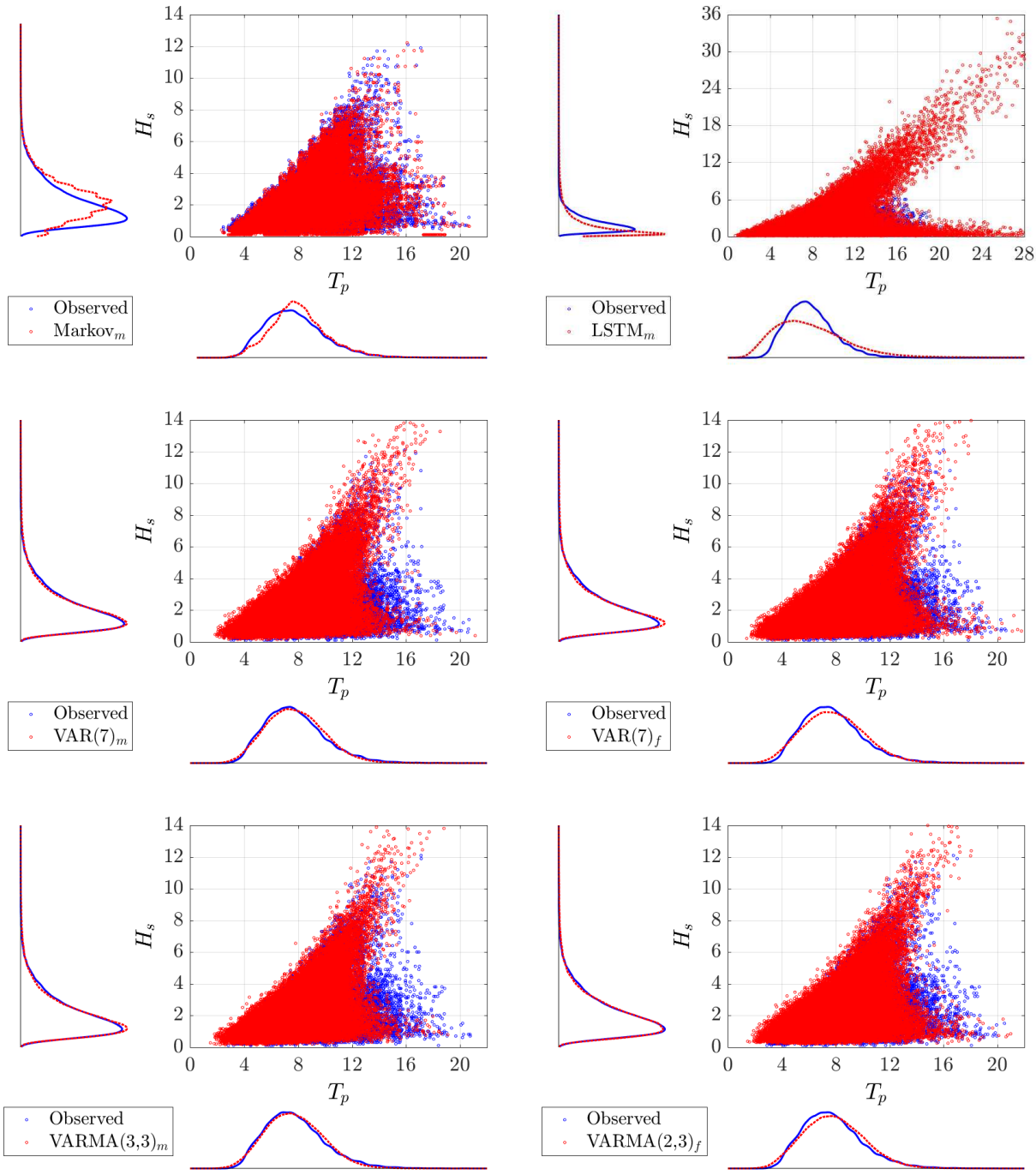


Figure B.3: Comparison Plot of Joint and Marginal Distribution for North Sea Series

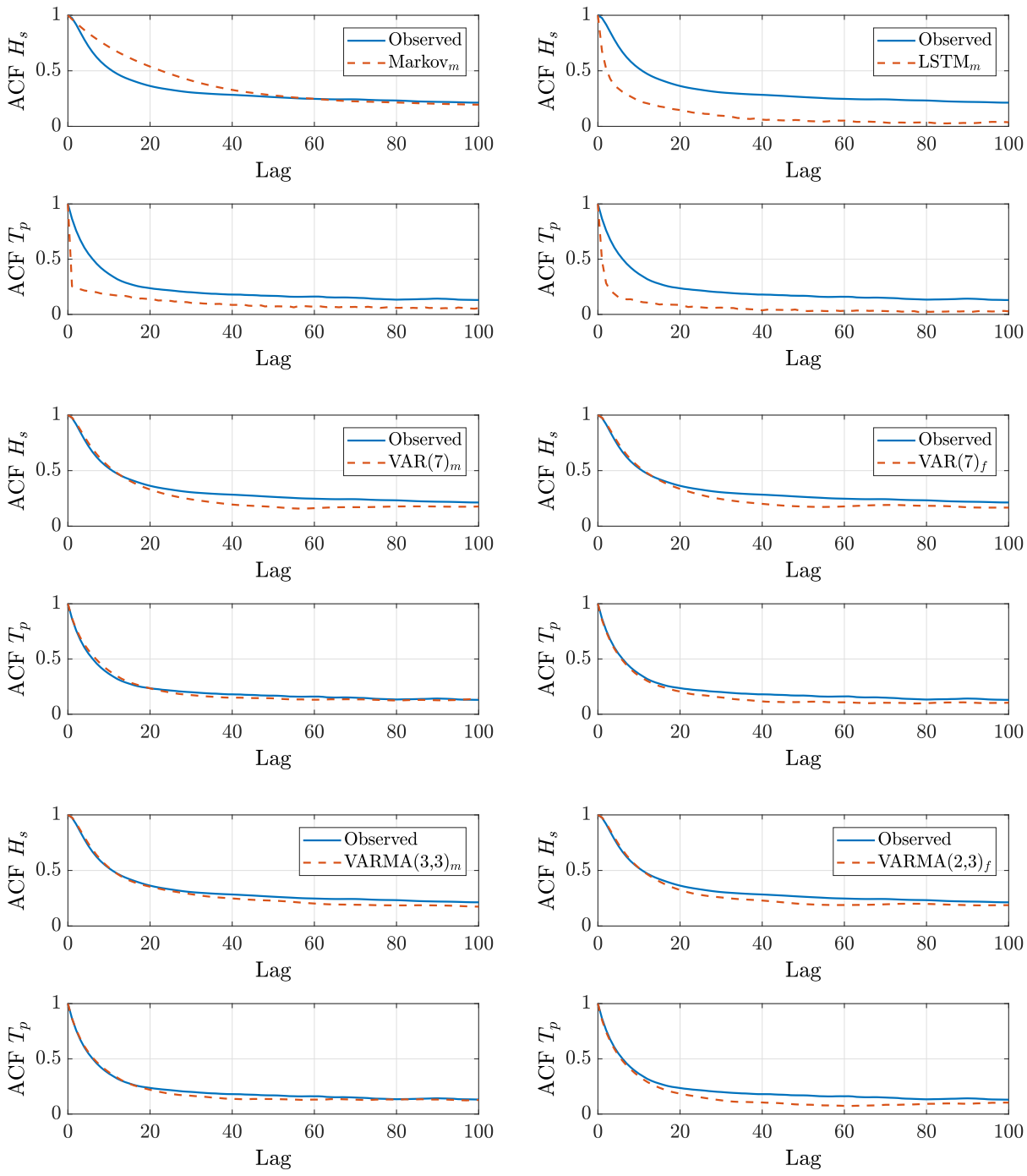


Figure B.4: Comparison Plot of Autocorrelation Function for North Sea Series

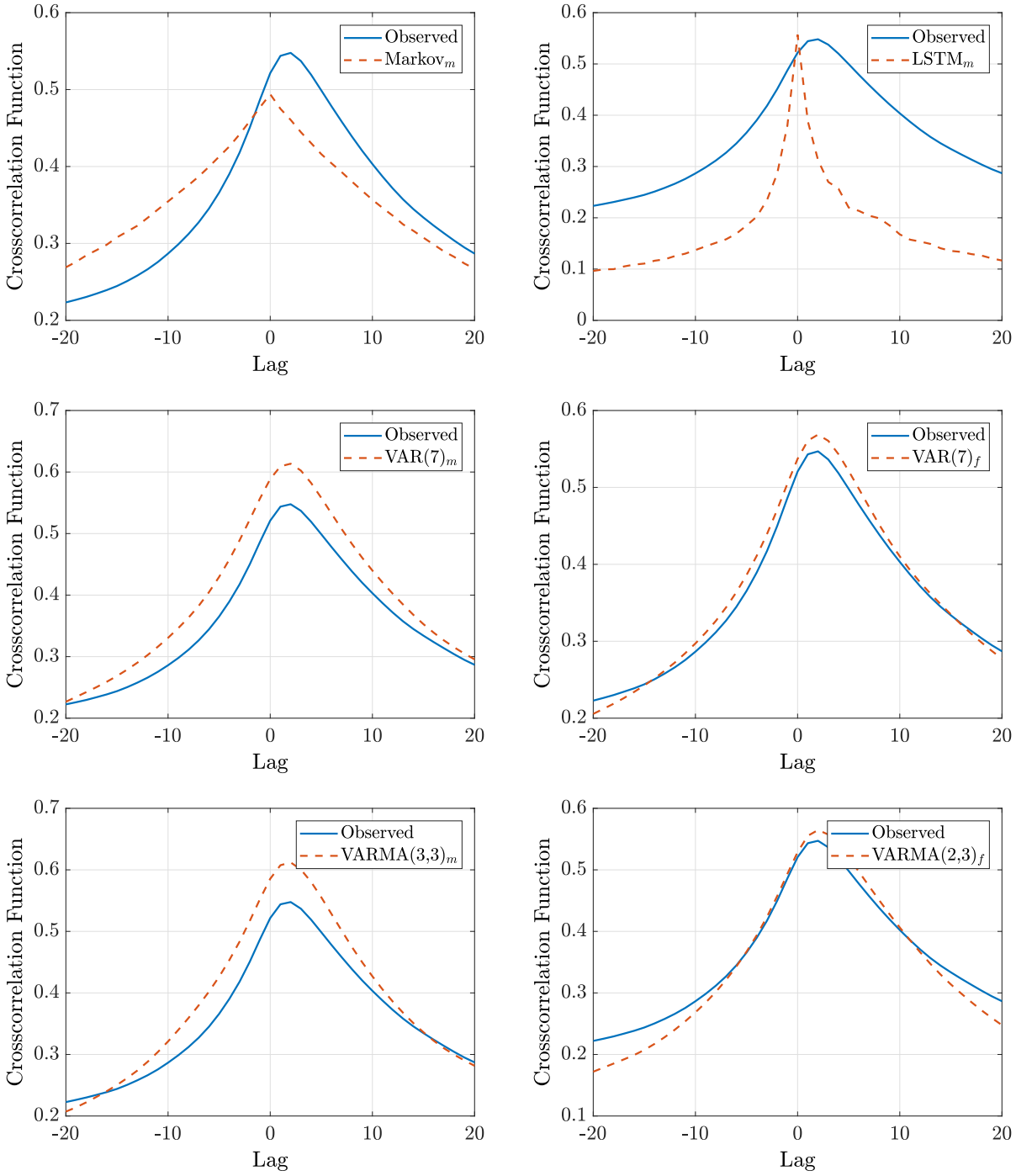


Figure B.5: Comparison Plot of Crosscorrelation Function for North Sea Series

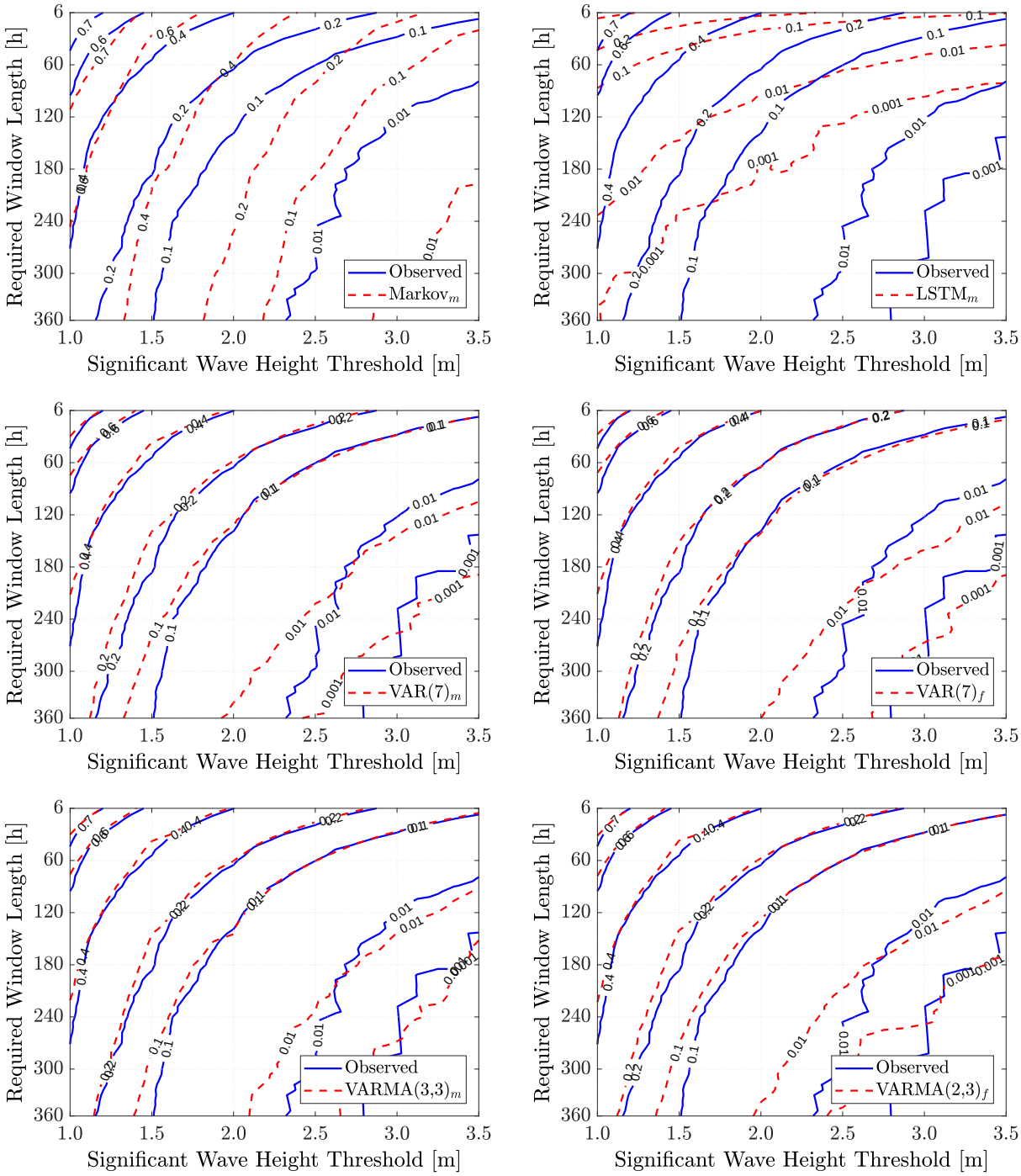


Figure B.6: Comparison Plot of Normalized Persistence Above Threshold Contours for North Sea Series

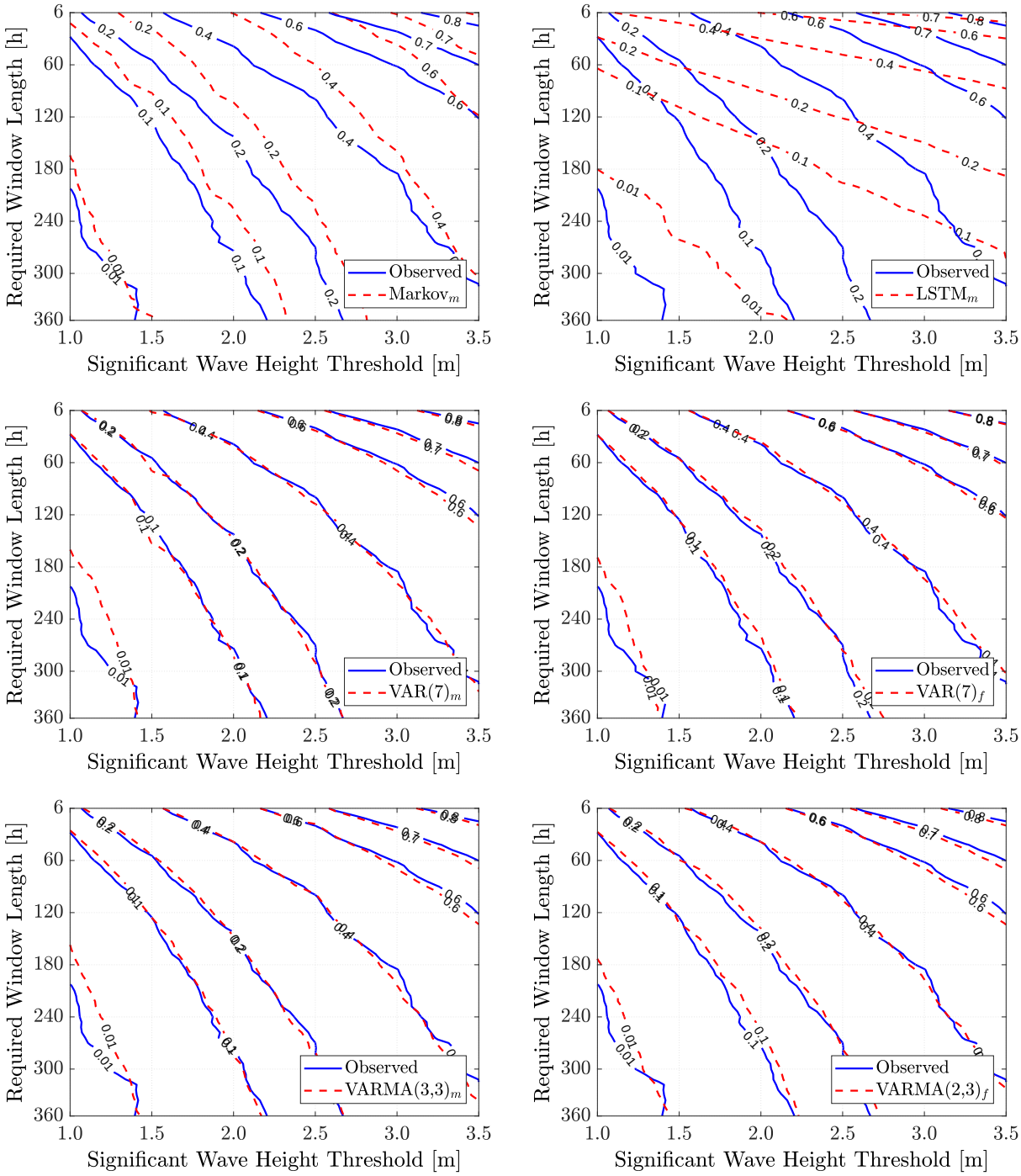


Figure B.7: Comparison Plot of Normalized Persistence Above Threshold Contours for North Sea Series

B.2 Mediterranean Sea Validation Results

Table B.2: Overall Lower Order Statistical Moments for Mediterranean Sea Series

Series	Mean H_s	Variance H_s	Mean T_p	Variance T_p	Covariance
Observed	1.267	0.999	6.122	2.947	1.414
Markov$_m$	1,403	0,990	6,424	2,924	1,391
LSTM Network$_m$	3.565	18.626	8.584	9.175	10.360
VAR(30)$_m$	1.287	1.374	6.128	3.074	1.628
VAR(10)$_f$	1.249	1.150	6.121	2.947	1.409
VARMA(3,3)$_m$	1.294	1.410	6.129	3.126	1.668
VARMA(3,3)$_f$	1.266	1.203	6.134	2.906	1.457

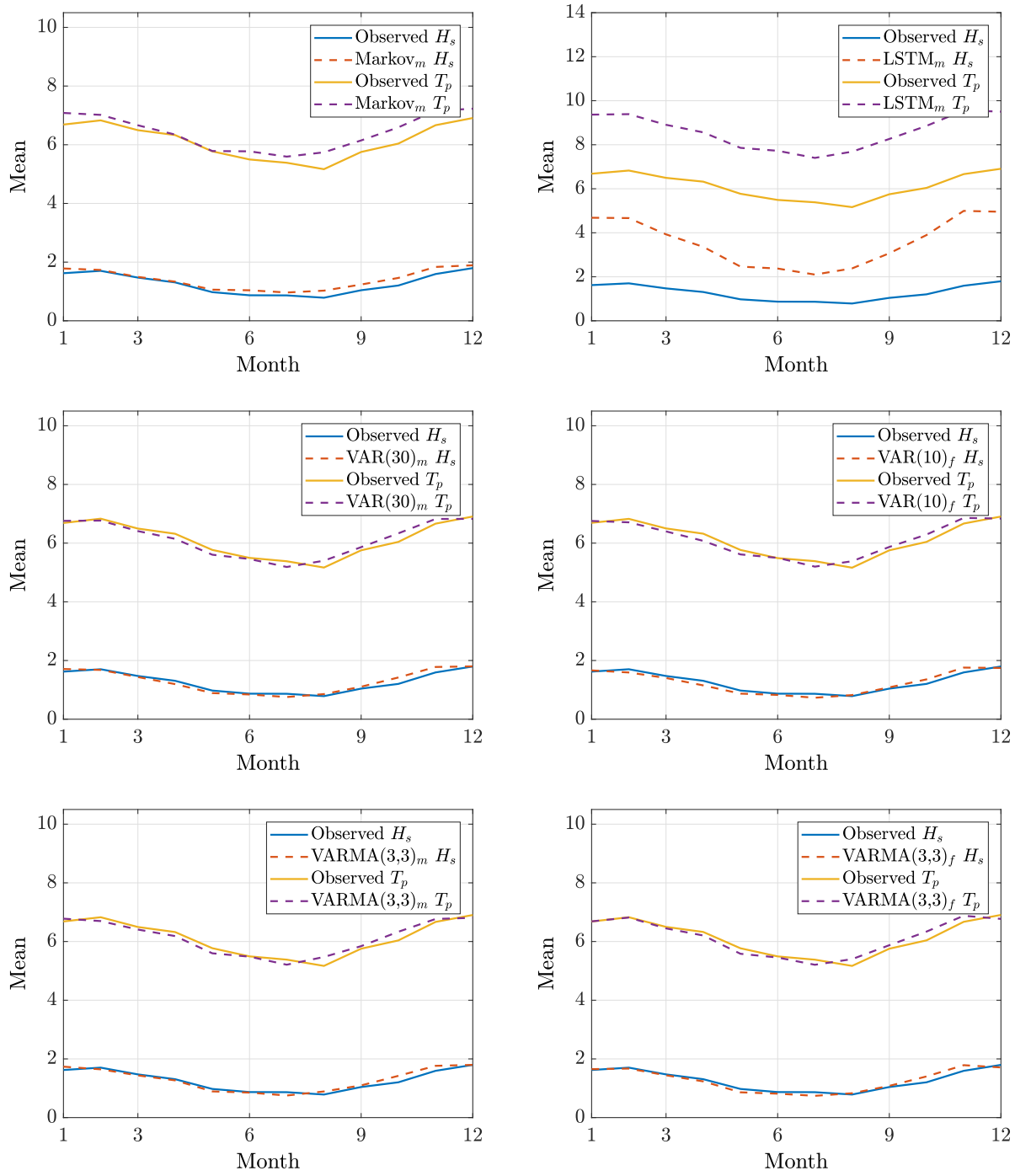


Figure B.8: Comparison Plot of Monthly Mean for Mediterranean Sea Series

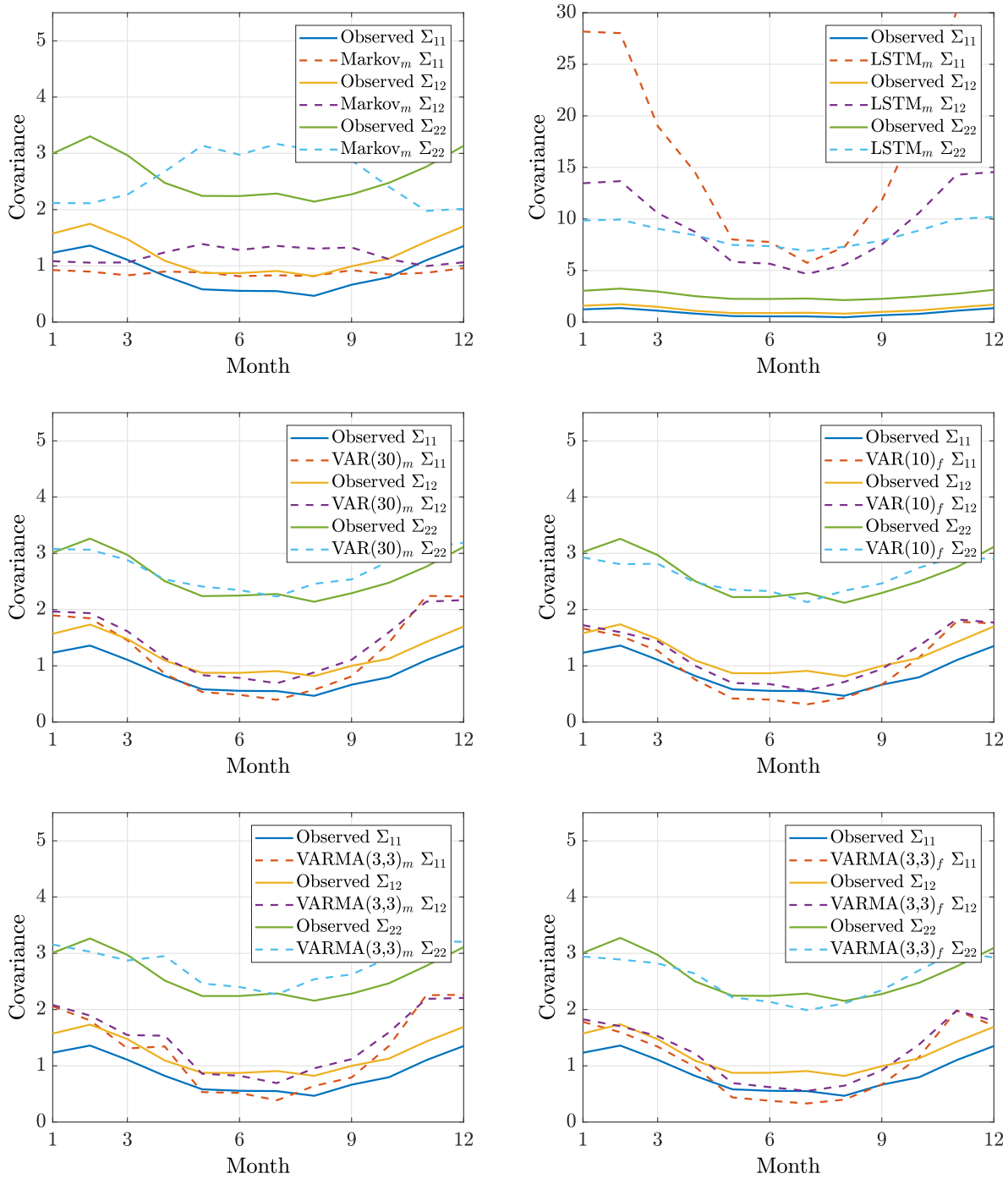


Figure B.9: Comparison Plot of Monthly Variances and Covariance for Mediterranean Sea Series

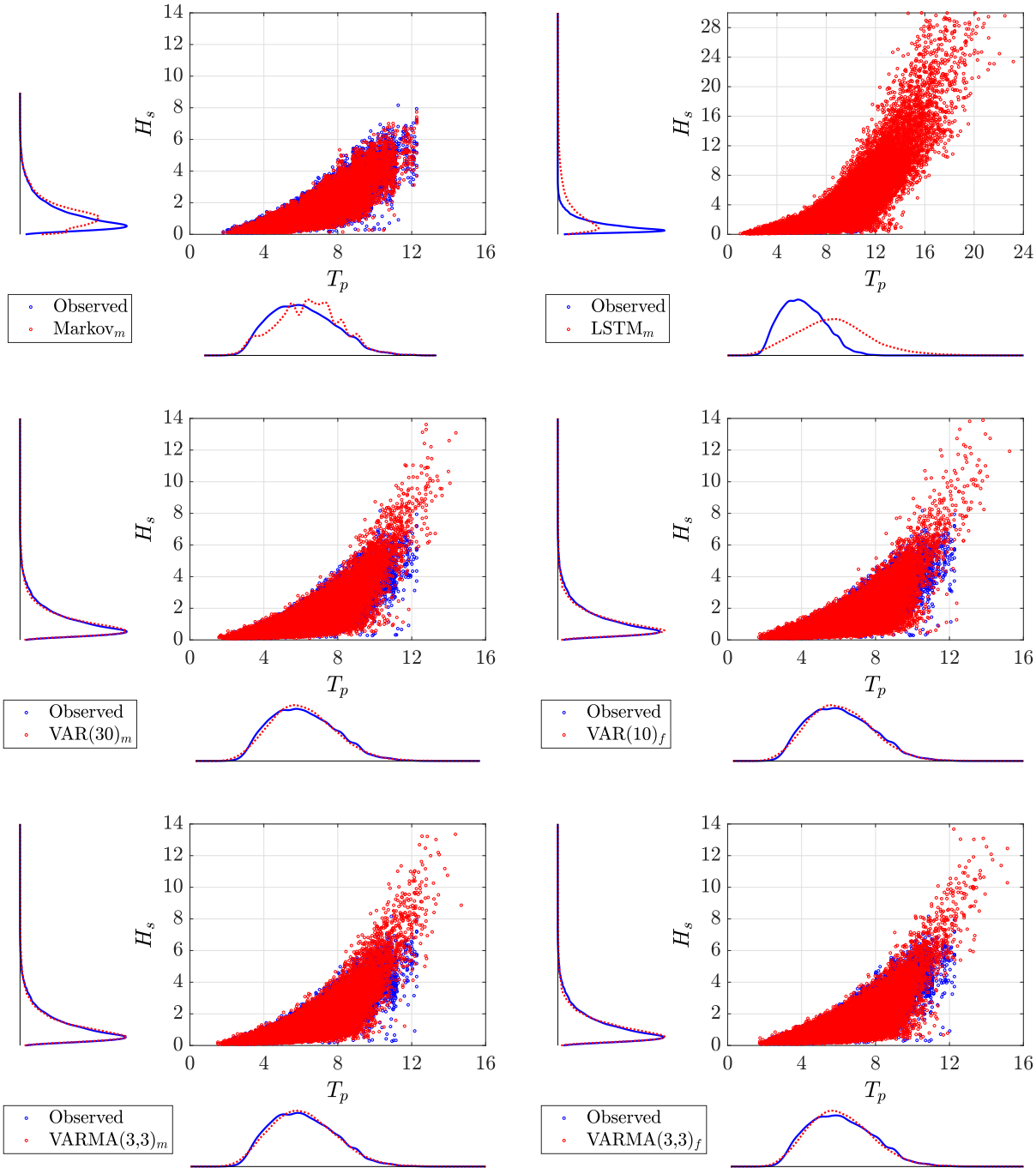


Figure B.10: Comparison Plot of Joint and Marginal Distribution for Mediterranean Sea Series

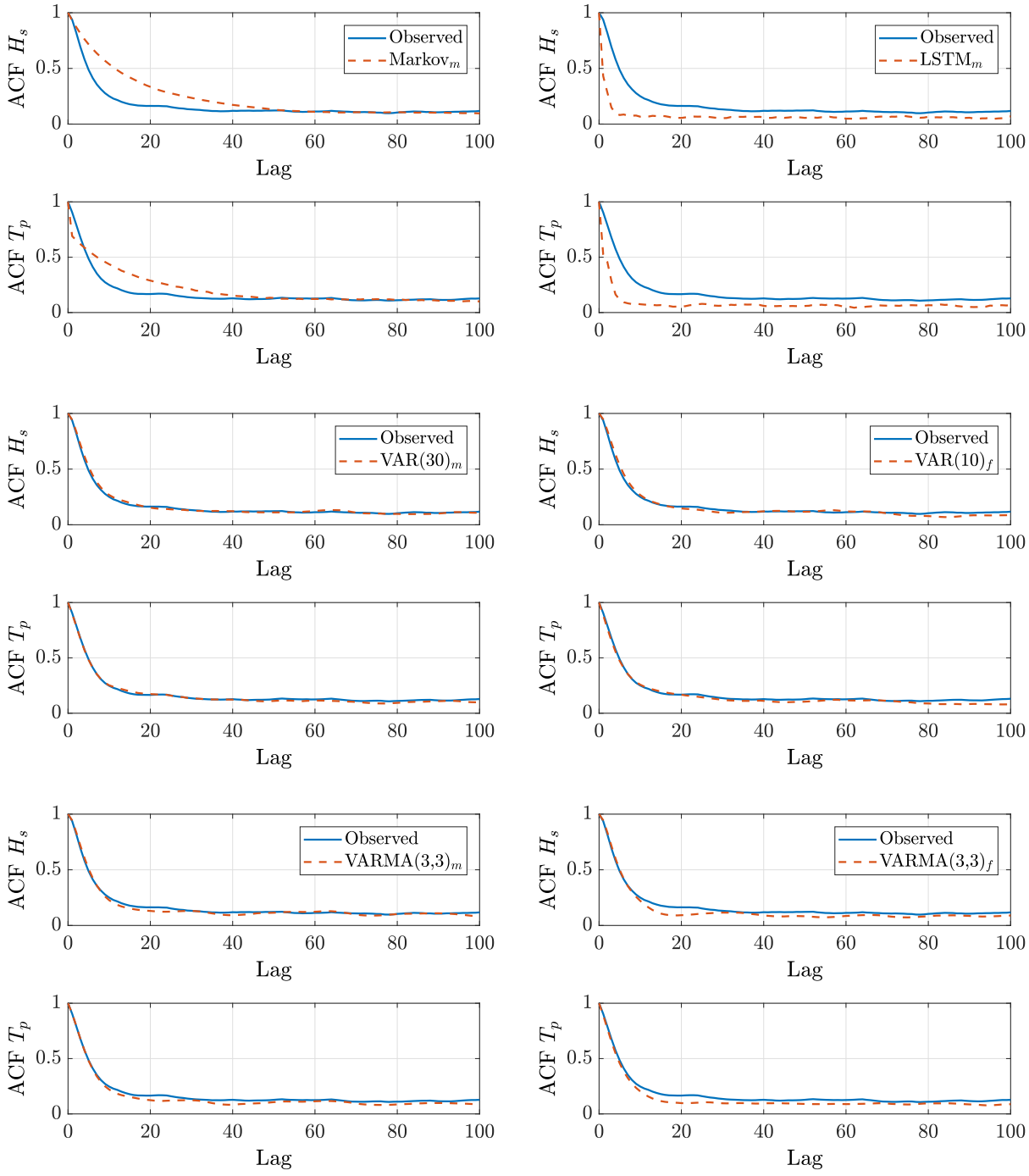


Figure B.11: Comparison Plot of Autocorrelation Function for Mediterranean Sea Series

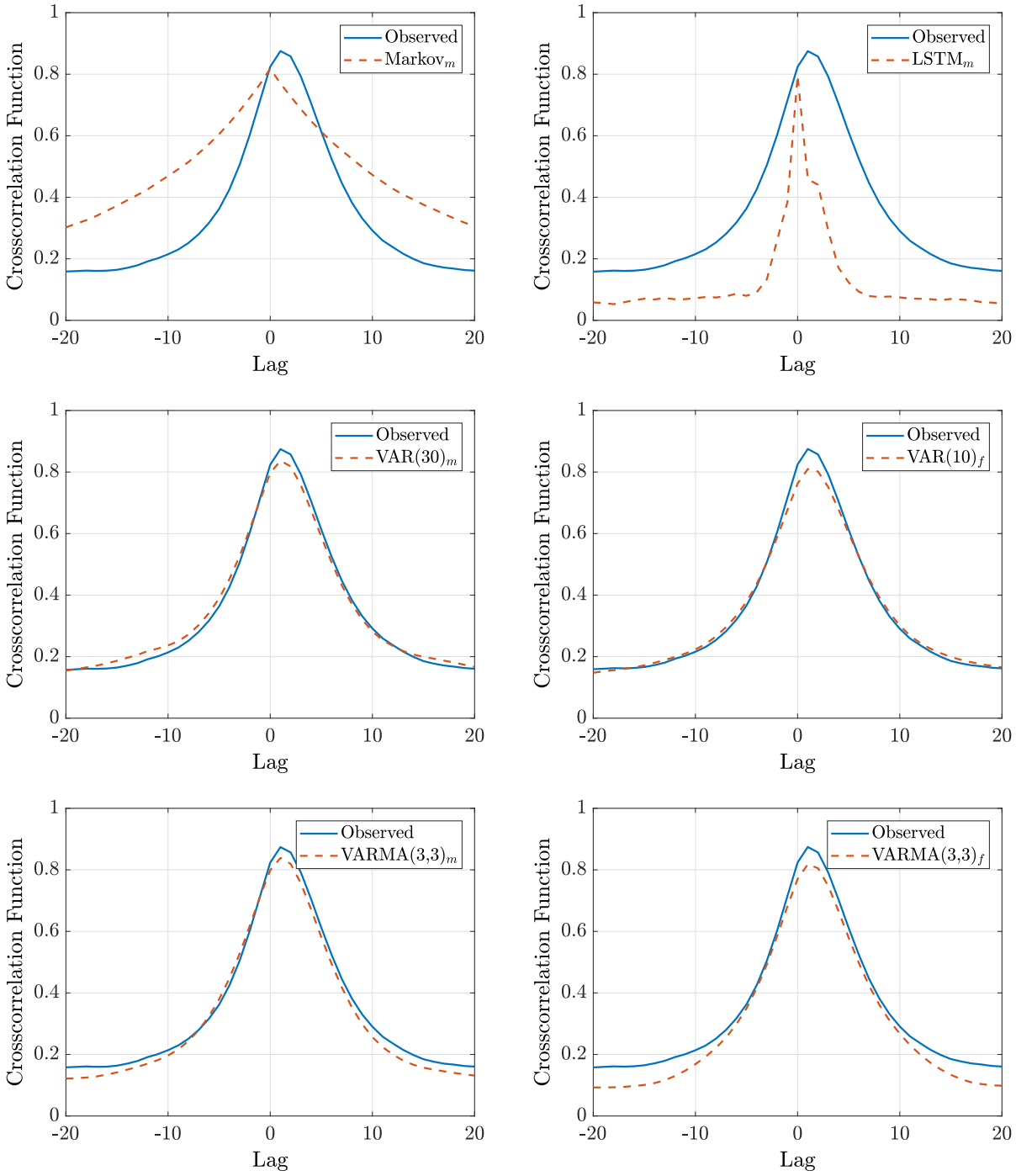


Figure B.12: Comparison Plot of Crosscorrelation Function for Mediterranean Sea Series

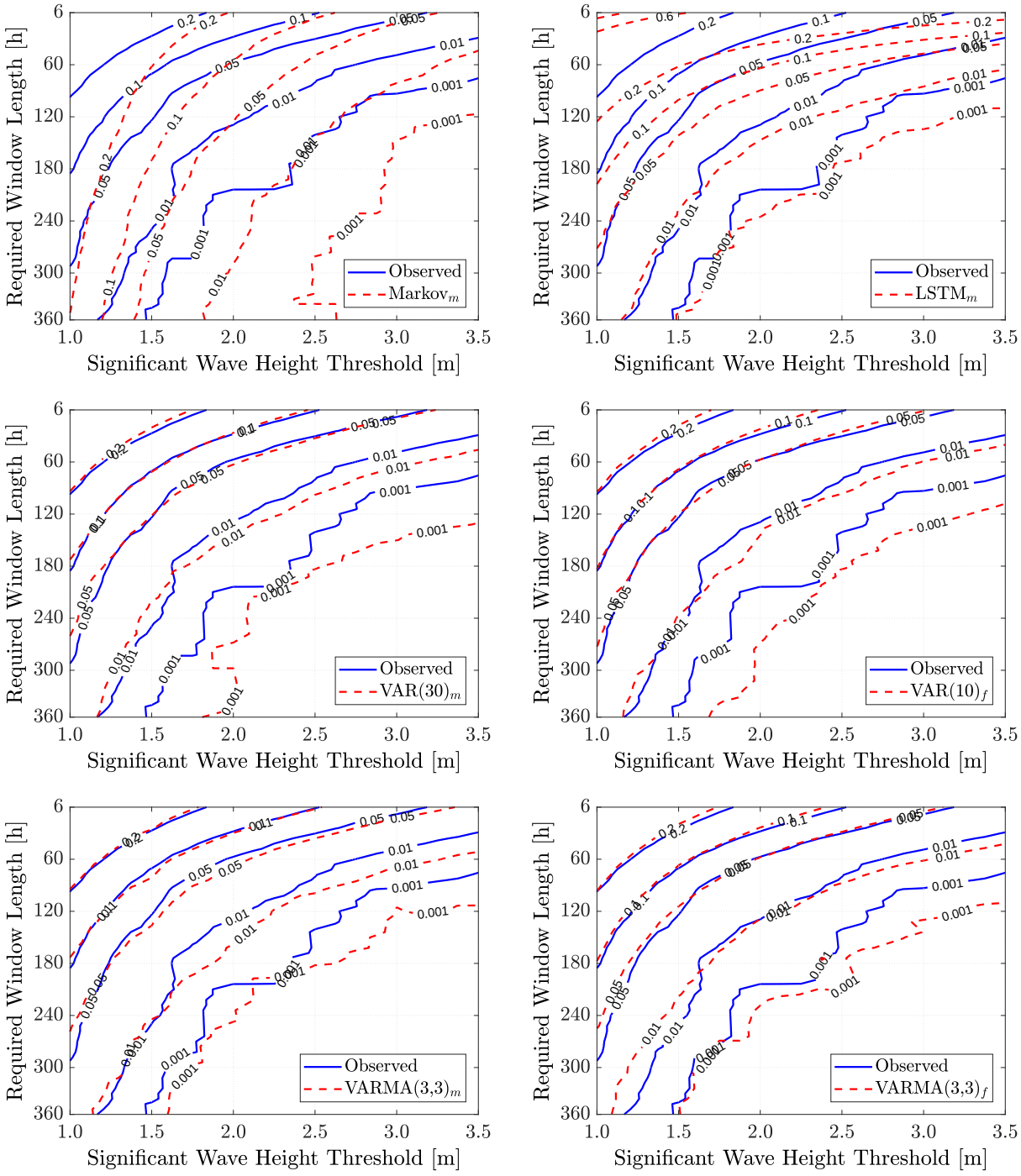


Figure B.13: Comparison Plot of Normalized Persistence Above Threshold Contours for Mediterranean Sea Series

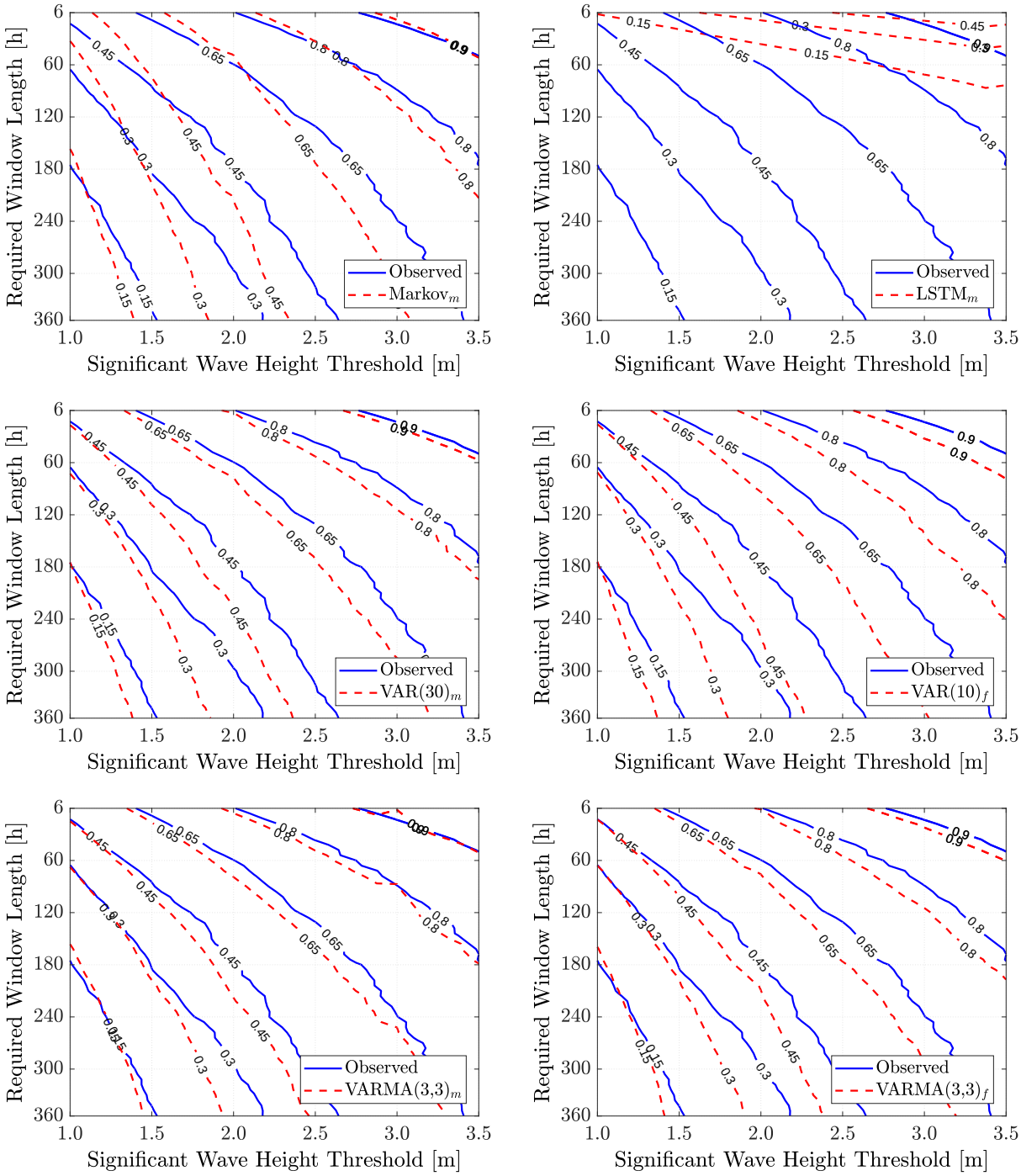


Figure B.14: Comparison Plot of Normalized Persistence Below Threshold Contours for North Sea Series

B.3 Box-Cox Comparison Results for VARMA Model on North Sea Series

Table B.3: Overall Lower Order Statistical Moments for Box-Cox and Rosenblatt Comparison

Model	Mean H_s	Variance H_s	Mean T_p	Variance T_p	Covariance
Observed	2.081	1.688	7.786	4.864	1.494
VARMA(3,1)$_{f,box}$	2,110	1,949	7,758	4,694	1,673
VARMA(2,3)$_f$	2.106	2.047	7.840	5.003	1.708

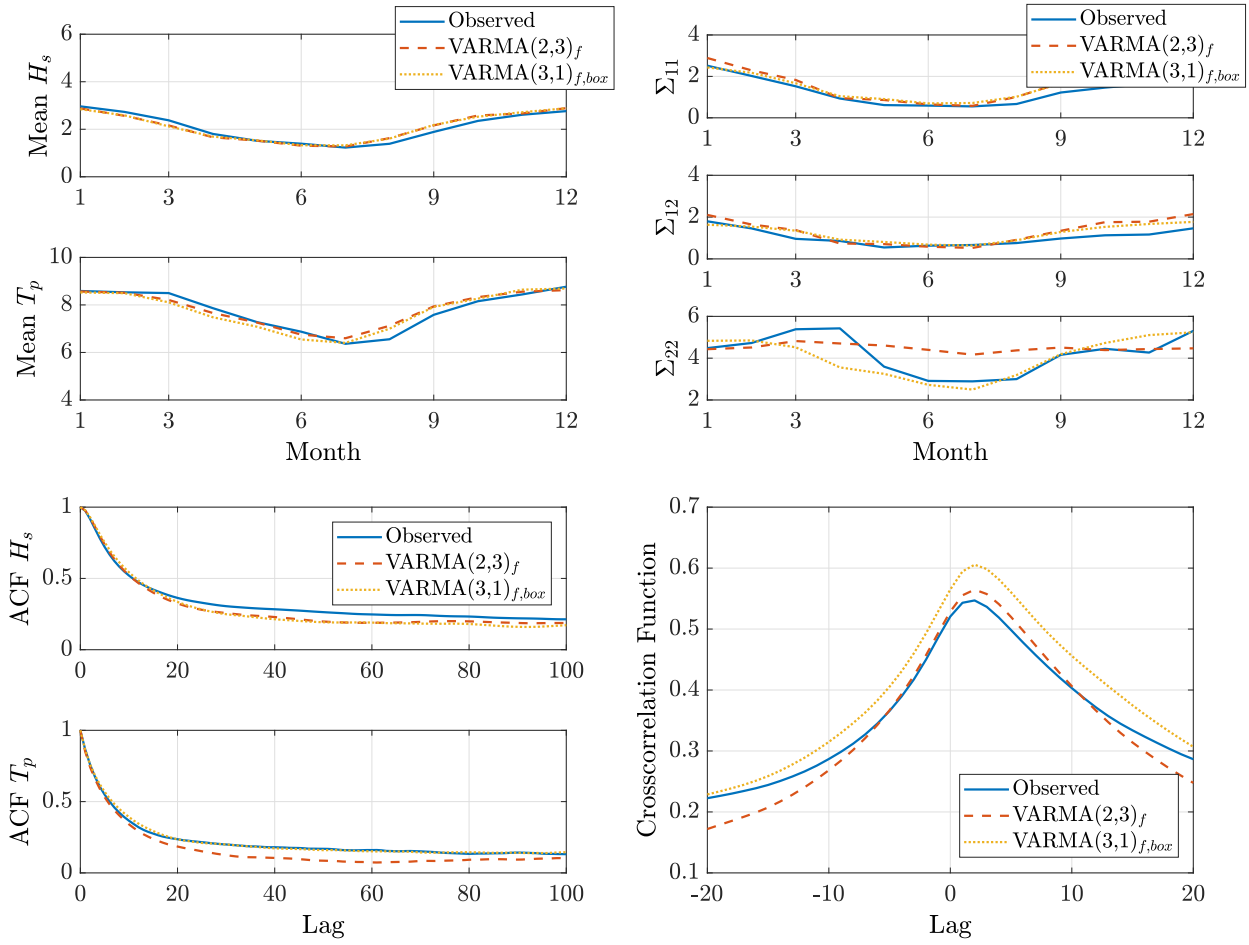


Figure B.15: Comparison Plots of VARMA models with Box-Cox and Lognormal Rosenblatt Transform - Part I

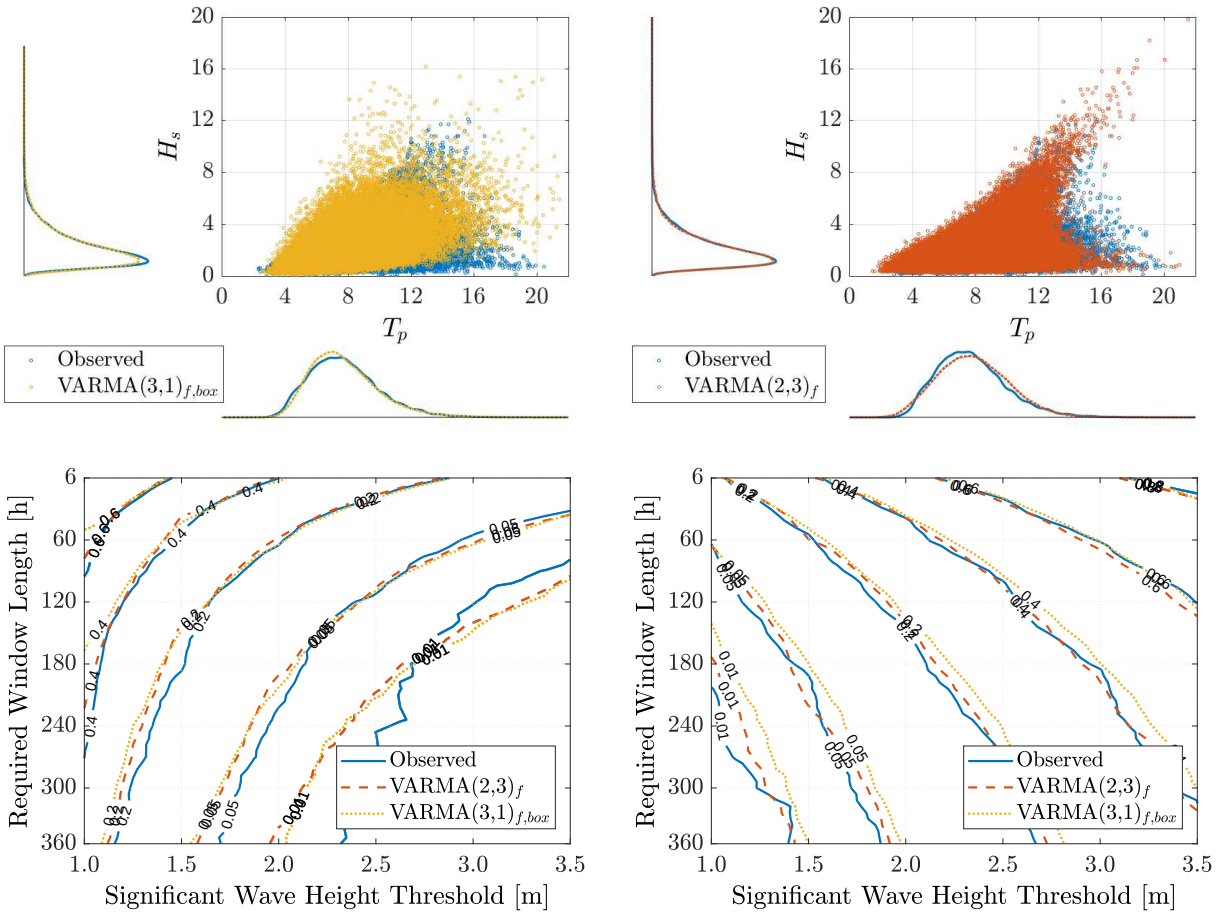


Figure B.16: Comparison Plots of VARMA models with Box-Cox and Lognormal Rosenblatt Transform - Part II

Appendix C

Code

C.1 Support Functions

LoadTimeSeries.m

```
% Function that returns Ts and Hs from time series from year X to year Y

function metoceanData = loadTimeSeries(startYear,endYear,filename)

fid = load(filename);

% Find start and end indexes
startIndex = find(fid(:,1) == startYear,1);
endIndex = find(fid(:,1) == endYear,1,'last');

% Return Hs and Tp with year and month indexes
metoceanData = [fid(startIndex:endIndex,1) fid(startIndex:endIndex,2) ...
    fid(startIndex:endIndex,7) fid(startIndex:endIndex,8)];
end
```

spread_calm_peak_period.m

```
% Spread calm weather observations
function [Tp_spread] = spread_calm_peak_period(metocean)

% Distriubute large first
cut_off_large = 100;

% Count number of occurrences of each unique Tp
unique_Tp = unique(metocean(:,4));
count_bin = zeros(length(unique_Tp),1);
for ii = 1:length(unique_Tp)
    count_bin(ii,1) = sum(metocean(:,4) == unique_Tp(ii));
end
```

```
% value of the clusters
cluster_val = unique_Tp(count_bin > cut_off_large);

% Merge close clusters
delta = 0.065; % 0.065
merge_list = zeros(length(cluster_val)-1,1);

% Identify clusters to merge
for ii = 1:length(cluster_val)-1
    if cluster_val(ii+1) - cluster_val(ii) < delta
        merge_list(ii) = 1;
        merge_list(ii+1) = 2;
    end
end

% Add zero for last element
merge_list(end+1) = 0;

% Vector holding target value for old clusters
merge_target1 = cluster_val(merge_list==1);
merge_target2 = cluster_val(merge_list==2);

merge_target = zeros(1,length(merge_list));

for ii = 0:(length(merge_target1)-1)
    merge_target(2*ii+1) = merge_target1(ii+1);
    merge_target(2*(ii+1)) = merge_target2(ii+1);
end

merged_new_value = zeros(1,length(merge_target));

% New values for clusters
counter = 1;
for ii = 1:length(merge_list)-1
    if merge_list(ii) == 1
        merged_new_value(counter) = (cluster_val(ii)+cluster_val(ii+1))/2;
        counter = counter + 1;
    elseif merge_list(ii) == 2
        merged_new_value(counter) = (cluster_val(ii)+cluster_val(ii-1))/2;
        counter = counter + 1;
    end

end

% Merge clusters
for ii = 1:length(metocean)
    for kk = 1:length(merge_target)
        if metocean(ii,4) == merge_target(kk)
            metocean(ii,4) = merged_new_value(kk);
            break
        end
    end
end
```

```
% Count number of occurrences of each unique Tp
unique_clustered_metocean = unique(metocean(:,4));
count_bin_cluster = zeros(length(unique_clustered_metocean),1);
for ii = 1:length(unique_clustered_metocean)
    count_bin_cluster(ii,1)=sum(metocean(:,4)==unique_clustered_metocean(ii));
end

cluster_val = unique_clustered_metocean(count_bin_cluster > cut_off_large);
cluster_size = count_bin_cluster(count_bin_cluster > cut_off_large);

% Unform distribution of the clusters using number of occurences as weights
for ii = 1:length(metocean(:,4))
    for jj = 2:(length(cluster_val)-1)
        if metocean(ii,4) == cluster_val(1)
            dist = cluster_val(2) - cluster_val(1);
            weight = cluster_val(2)/cluster_val(1);
            if rand < weight
                metocean(ii,4) = metocean(ii,4) + dist*rand;
            else
                metocean(ii,4) = metocean(ii,4) - dist*rand;
            end
            break
        elseif metocean(ii,4) == cluster_val(jj)
            dist_1 = cluster_val(jj)-cluster_val(jj-1);
            dist_2 = cluster_val(jj+1)-cluster_val(jj);
            weight = cluster_size(jj-1)/(cluster_size(jj-1)+cluster_size(jj+1));
            if rand < weight
                metocean(ii,4) = cluster_val(jj) - dist_1*rand;
            else
                metocean(ii,4) = cluster_val(jj) + dist_2*rand;
            end
            break
        elseif metocean(ii,4) == cluster_val(end)
            dist = 2*(cluster_val(end) - cluster_val(end-1));
            metocean(ii,4) = cluster_val(end-1) + dist*rand;
            break
        end
    end
end

% Distibute remaining small clusters
cluster_val = unique_clustered_metocean(count_bin_cluster < cut_off_large ...
& count_bin_cluster > 40);
cluster_size = count_bin_cluster(count_bin_cluster < cut_off_large ...
& count_bin_cluster > 40);


for ii = 1:length(metocean(:,4))
    for jj = 2:(length(cluster_val)-1)
        if metocean(ii,4) == cluster_val(1)
            dist = cluster_val(2) - cluster_val(1);
            weight = cluster_val(2)/cluster_val(1);
            if rand < weight
```

```
                metocean(ii,4) = metocean(ii,4) + dist*rand;
            else
                metocean(ii,4) = metocean(ii,4) - dist*rand;
            end
            break
        elseif metocean(ii,4) == cluster_val(jj)
            dist_1 = cluster_val(jj)-cluster_val(jj-1);
            dist_2 = cluster_val(jj+1)-cluster_val(jj);
            weight = cluster_size(jj-1)/(cluster_size(jj-1)+cluster_size(jj+1));
            if rand < weight
                metocean(ii,4) = cluster_val(jj) - dist_1*rand;
            else
                metocean(ii,4) = cluster_val(jj) + dist_2*rand;
            end
            break
        elseif metocean(ii,4) == cluster_val(end)
            dist = 2*(cluster_val(end) - cluster_val(end-1));
            metocean(ii,4) = cluster_val(end-1) + dist*rand;
            break
        end
    end
end

Tp_spread = metocean(:,4);
end
```

seasonal_trans.m

```
function [yearMeanFit,yearCovFit,midMonthHours] = seasonal_trans(startYear,...
endYear,metoceanBox,year_length)
% Calculate the points necessary to fit Fourier approximations for
% covariance and mean

% Number of years
nYears = endYear-startYear+1;
% Number of months
nMonths = 12*nYears;

% Initilize
monthHsMean = zeros(1,nMonths); monthTpMean = zeros(1,nMonths);
monthCov = zeros(2,2,nMonths);
counter = 1; % Set start month to 1

%% Monthly mean and standard deviation
% Build matrix of mean and standard deviation for each month in the series
for ii = startYear:1:endYear
    for jj = 1:12
        monthHsMean(counter) = mean(metoceanBox(metoceanBox(:,1) == ii ...
                & metoceanBox(:,2) == jj,3));
        monthTpMean(counter) = mean(metoceanBox(metoceanBox(:,1) == ii ...
                & metoceanBox(:,2) == jj,4));
```

```
        monthCov(:,:,counter) = cov(metoceanBox(metoceanBox(:,1) == ii ...
                & metoceanBox(:,2) == jj,3), ...
            metoceanBox(metoceanBox(:,1) == ii & metoceanBox(:,2) == jj,4));
        counter = counter + 1;
    end
end

%% Yearly Mean and Standard Deviation
% Build matrix of mean and standard deviation for the average of each month
% over all years in the series

% Initilize
yearHsMeanSum = zeros(1,12); yearTpMeanSum = zeros(1,12);
yearCovSum = zeros(2,2,12);

% Yearly mean
% Sum for each month over all years
for ii = 1:12
    for jj = 1:nYears
        yearHsMeanSum(ii) = yearHsMeanSum(ii) + monthHsMean(ii+12*(jj-1));
        yearTpMeanSum(ii) = yearTpMeanSum(ii) + monthTpMean(ii+12*(jj-1));
        yearCovSum(:,:,ii) = yearCovSum(:,:,ii) + monthCov(:,:,ii+12*(jj-1));
    end
end

% Divide by number of years to find average
yearTpMean = yearTpMeanSum/nYears;
yearHsMean = yearHsMeanSum/nYears;
yearCov = yearCovSum/nYears;

% Combine mean values
yearMean = transpose([yearHsMean;yearTpMean]);

%% Seasonal Transformation
% Mean periodic function M

% Vector containing the middle day of each month in hours from year start
% Day 1 = 365, such that the period is equal to one year
midMonthHours = transpose([1 15.5 45 74.5 105 135.5 166 196.5 227.5 ...
258 288.5 319 349.5 365]*(year_length/365));

% Row containing the average mean value for the 1st and 365th day
% Found as the mean of month 1 and month 12
startEndDay(1,:) = (yearMean(1,:) + yearMean(end,:))/2;

% Add the average value row as the first and last row in month mean matrix
yearMeanFit = [startEndDay;yearMean;startEndDay];



% Covariance periodic function SIGMA

% Initialize
S11 = zeros(12,1);S12 = zeros(12,1);S22 = zeros(12,1);
```

```
% Build vector for each covariance per month
for ii = 1:12
    S11(ii,1) = yearCov(1,1,ii);
    S12(ii,1) = yearCov(1,2,ii); % S12 = S21
    S22(ii,1) = yearCov(2,2,ii);
end

% Combine monthly values of covariance into monthly covariance matrix
obsMonthCov = [S11 S12 S12 S22];

% Row containing the average mean value for the 1st and 365th day
% Found as the mean of month 1 and month 12
startEndDay2(1,:) = (obsMonthCov(1,:) + obsMonthCov(end,:))/2;

% Add the average value row as the first and last row in month mean matrix
yearCovFit = [startEndDay2;obsMonthCov;startEndDay2];

end
```

scatterDiagram.m

```
%------- creating scatter diagram using archived Hs and randomized Tp----------

% This function was used with permission from its author Endre Sandvik.

function scatterD = scatterDiagram(Hs, Tp)
%%
%Defining different variables used in the program
n = length(Hs);
a = 41;
b = 27;

%Creating scatter diagram matrix
scatterD = zeros(a,b);
scatterD(1:39,1) = 0:0.5:19;
scatterD(1,1:25) = 0:1:24;

%%
%For-loop for sorting data into classes
for i = 1:n
        %Sorting Hs into classes of 0.5
    diff = Hs(i) - floor(Hs(i));
    if diff >= 0.75
        Hs(i)=ceil(Hs(i));
    elseif diff >= 0.25 && diff < 0.75

        Hs(i) = floor(Hs(i)) + 0.5;
    else
        Hs(i) = max([floor(Hs(i)) 0.5]);
```

```
    end
    %Sorting Tp into classes of 1
    diffTp = Tp(i) - floor(Tp(i));
    if diffTp >= 0.5
        Tp(i)=ceil(Tp(i));
    else
        Tp(i) = max([floor(Tp(i)) 1]);
    end
end
%%
for i = 1:n
    if Hs(i) < 19 && Tp(i) < 24
   %Assigning the sea state to the correct location in the scatter diagram
   scatterD(1+2*Hs(i),1+Tp(i)) = scatterD(1+2*Hs(i),1+Tp(i)) + 1;

   %Adding 1 to the count of Hs
   scatterD(1+2*Hs(i),b-1) = scatterD(1+2*Hs(i),b-1) + 1;

   %Adding 1 to the count of Tp
   scatterD(a-1,1+Tp(i)) = scatterD(a-1,1+Tp(i)) + 1;
    end

end
%%
%Counting cumulative i-values
for i = 2:a-1
    scatterD(i,b) = scatterD(i-1,b) + scatterD(i,b-1);
end
for i = 2:b-1
    scatterD(a,i) = scatterD(a,i-1) + scatterD(a-1,i);
end
%%
end
```

meanAndStdDev.m

```
% ------- fitting a curve to exspected value and variance -------

% Used with permission from its author Endre Sandvik
function [a, b] = meanAndStdDev(scatterD,plotOn)

%Function for establishing coefficients a1,a2,a3 in function for ln(expected
%value) and b1,b2,b3 in function for ln(variance).

%Length of Hs and Tp classes found from scatterdiagram
k = length(scatterD(1,:)); %Hs
d = length(scatterD(:,1)); %Tp

%Declearing vectors for expected value and variance
my = zeros(1,d-3);
variance = zeros(1,d-3);
```

```
%%
%Estimating variance and expected value from scatterdiagram

data_vec = [];
h_var = [];
for i = 2:d-2%Hs
    for j = 2:k-2%Tp

        if scatterD(i,j) == 0

        else
           data_vec = [data_vec (j-1)*ones(1,scatterD(i,j))];
        end

    end

    if length(data_vec) >= 1
        param = lognfit(data_vec);
        my(i-1) = param(1);
        variance(i-1) = param(2)^2;
        data_vec = [];
        h_var = [h_var i-1];
    end
end


%%
%Fitting curve to values of expected value and variance using least square
%curve fitting method

%f1 is the type of function to be fitted to the expected values
f1 = @(a,adata) a(1)+a(2)*adata.^a(3);
%Fitting the curve (f1) to the expected values
a = lsqcurvefit(f1,[0.001 0.001 0.001],scatterD(h_var+1,1)',my(h_var));

%f2 is the type of function to be fitted to the variances
f2 = @(b,bdata) b(1)+b(2)*exp(-bdata*b(3));
%Fitting the curve (f2) to the variances
b = lsqcurvefit(f2,[0.001 0.001 0.001],scatterD(h_var+1,1)',variance(h_var),...
                [0 0 0],[3 3 3]);

%%

if plotOn == 1
    red = [0.8500, 0.3250, 0.0980];

    %Plot of empirical values from data for E[lnTp|Hs] and fitted curve for
    set(figure,'name','Curve fitting expected value','numbertitle','off')
    scatter(scatterD(h_var+1,1),my(h_var),'Marker','x','LineWidth',2);
    hold on
    plot(scatterD(h_var+1,1),f1(a,scatterD(h_var+1,1)),'LineStyle','--',...
                    'Color',red,'LineWidth',1.5)
    xlabel('$H_{s}$','FontSize',18)
    ylabel('$\mu_{ln T_p(H_s)}$','FontSize',18)
```

```
    legend('Calculated Values','Fitted curve','Location','northwest')
    yticks(1.5:0.25:2.5)
    grid on

    %Plot of empirical ln(variance) vs fitted curve
    set(figure,'name','Curve fitting variance','numbertitle','off')
    scatter(scatterD(h_var+1,1),variance(h_var),'Marker','x','LineWidth',2);
    hold on
    plot(scatterD(h_var+1,1),f2(b,scatterD(h_var+1,1)),'LineStyle','--',...
        'Color',red,'LineWidth',1.5)
    xlabel('$H_{s}$','FontSize',18)
    ylabel('$\sigma^2_{ln T_p(H_s)}$','FontSize',18)
    legend('Calculated Values','Fitted Curve','Location','northeast')
    grid on
end

end
```

boxcoxn.m

```
% Estimate parameters lamba for multivariate BoxCox transformation

% From: https://stackoverflow.com/questions/15501455/box-cox-transformation
%-for-multivariate-normality-in-matlab?rq=1 by user T.Lan

function lambda=boxcoxn(x)
[m,n]=size(x);
lambda_ini=zeros(n,1);
for ii=1:n
    [temp,lambda_ini(ii,1)]=boxcox(x(:,ii));
end
fun=@(lambda)(log(det((cov(((x.^repmat(lambda',m,1)-1)./repmat(lambda',...
    m,1))))))*m/2-(lambda-1)'*(sum(log(x)))');
lambda=fminsearch(fun,lambda_ini);
end
```

C.2 Residual Transformation Scripts

generate_residuals.m

```
% Generate Residuals
clear all
clc

% Parameters
start_year = 1988; % Start year of time series
end_year = 2015; % Final year of time series, including this year
```

```
filename = 'WAM10.txt'; % WAM 10 Metocean file without headers.
harsh = 1; % Use any value for calm.
rosen = 1; %

% Load Hs, Tp, year indices and month indices from start year to and
% including end year
if harsh == 1
    metocean = loadTimeSeries(start_year,end_year,filename);
    year_length = 2920;
else
    metocean = csvread(filename);
    year_length = 1460;
end

% Length of time series
series_length = length(metocean(:,1));


%% Uniformly distribute Tp
if harsh == 1
    % Spread Tp with uniform distribution
    Tp_spread = zeros(length(metocean),1);
    for ii = 1:length(metocean(:,4))
        n = round(1+log(metocean(ii,4)/3.244)/0.09525);
        metocean(ii,4) = 3.244*exp(0.09525*(n-0.5-rand));
    end
else
    metocean(:,4) = spread_calm_peak_period(metocean);
end

%% Transformation to normality
% Allocate year and month indices
metocean_norm(:,1) = metocean(:,1);
metocean_norm(:,2) = metocean(:,2);

if rosen == 1
    % Rosenblatt transform
    %Creating scatter diagram of
    scatter_diagram = scatterDiagram(metocean(:,3),metocean(:,4));

    % Determine regression coefficients for mu and sigma for lognormal
    [a, b] = meanAndStdDev(scatter_diagram,0);

    % Fit lognormal distribution to Hs and find parameters mu and sigma
    logn_par = lognfit(metocean(:,3));

    % Transform Hs to lognormal probability
    F_Hs = logncdf(metocean(:,3),logn_par(1),logn_par(2));

    % Transform Tp to lognormal probability conditioned on Hs
    F_Tp = zeros(length(metocean(:,3)),1);
    for ii = 1:length(metocean(:,3))
        % Conditioned mu and sigma values
        mu = a(1) + a(2)*(metocean(ii,3)^a(3));
```

```
        sigma = sqrt(b(1) + b(2)*exp(-b(3)*metocean(ii,3)));
        F_Tp(ii) = logncdf(metocean(ii,4),mu,sigma);
    end
    % Inverse standard normal trasnform of Hs and Tp
    [metocean_norm(:,3)] = norminv(F_Hs);
    [metocean_norm(:,4)] = norminv(F_Tp);
else
    % Multivariate BoxCox transform
    lambda_cox = boxcoxn(metocean(:,3:4));
    [metocean_norm(:,3)] = boxcox(lambda_cox(1),metocean(:,3));
    [metocean_norm(:,4)]= boxcox(lambda_cox(2),metocean(:,4));
end

%% Seasonal Transformation
[yearMeanFit,yearCovFit,midMonthHours] = seasonal_trans(start_year,...
end_year,metocean_norm,year_length);

% Fit Fourier series to monthly mean and covariance
M_Hs = fit(midMonthHours,yearMeanFit(:,1),'fourier6'); % M Hs
M_Tp = fit(midMonthHours,yearMeanFit(:,2),'fourier6'); % M Tp
S_11 = fit(midMonthHours,yearCovFit(:,1),'fourier6');
S_12 = fit(midMonthHours,yearCovFit(:,2),'fourier6'); % S12=S21
S_22 = fit(midMonthHours,yearCovFit(:,4),'fourier6');

%% Generate Residuals
% Pre-allocate
W = zeros(series_length,2);

% Perform seasonal transformation W = (Y-M)/S
for ii = 1:series_length
    Y = [metocean_norm(ii,3); metocean_norm(ii,4)];
    M = [M_Hs(ii); M_Tp(ii)];
    S = sqrt([S_11(ii) 0; 0 S_22(ii)]);
    %W(ii,:)=(Y-M);
    W(ii,:)=S\(Y-M); % x = A\b --- Ax = b
end

% Write residuals W to .csv file
out = [metocean(:,1:2) W];

%out = out(out(:,1) == 1989,:);
writeFileName = 'residuals_pred_test.csv';
csvwrite(writeFileName,out)

%% Validation Plots
% Plot of Fourier approximations for validation of fit
figure(1)
subplot(1,2,1)
plot(M_Hs,midMonthHours,yearMeanFit(:,1))
title('M Hs')
subplot(1,2,2)
plot(M_Tp,midMonthHours,yearMeanFit(:,2))
title('M Tp')
```

```
figure(2)
plot(S_11,midMonthHours,yearCovFit(:,1))
hold on
plot(S_12,midMonthHours,yearCovFit(:,2))
hold on
plot(S_22,midMonthHours,yearCovFit(:,4))
hold off

if rosen == 1
    figure(3)
    subplot(1,2,1)
    histogram(norminv(F_Hs))
    subplot(1,2,2)
    histogram(norminv(F_Tp))

    figure(4)
    scatter(norminv(F_Hs),norminv(F_Tp))
    xlabel('u1')
    ylabel('u2')
    title('Residuals')
end
```

residuals_backtrans.m

```
% Back-transform residuals

clear all

% Parameters
start_year = 1990; % Start year of time series
end_year = 2015; % Final year of time series, including this year
observed_filename = 'WAM10.txt'; % WAM 10 Metocean file without headers.
residual_filename = 'Residuals_VARMA_31_BOXCOX_MS_H.csv';
num_rep = 10;
harsh = 1;
std_on = 1;
write = 0;
rosen = 0;

% Load Hs, Tp, year indices and month indices from start year to and
% including end year
if harsh == 1
    metocean = loadTimeSeries(start_year,end_year,observed_filename);
    year_length = 2920;
else
    metocean = csvread(observed_filename);
    year_length = 1460;
end

% Load residual series
W_sim_raw = load(residual_filename);
```

```
% Length of time series
series_length = length(W_sim_raw);

% Allocate year and month indices
metocean_norm(:,1) = metocean(:,1);
metocean_norm(:,2) = metocean(:,2);

%% Uniformly distribute Tp
if harsh == 1
    % Spread Tp with uniform distribution
    Tp_spread = zeros(length(metocean),1);
    for ii = 1:length(metocean(:,4))
        n = round(1+log(metocean(ii,4)/3.244)/0.09525);
        metocean(ii,4) = 3.244*exp(0.09525*(n-0.5-rand));
    end
else
    metocean(:,4) = spread_calm_peak_period(metocean);
end

%% Transformation to normality
% Allocate year and month indices
metocean_norm(:,1) = metocean(:,1);
metocean_norm(:,2) = metocean(:,2);

if rosen == 1
    % Rosenblatt transform
    %Creating scatter diagram of
    scatter_diagram = scatterDiagram(metocean(:,3),metocean(:,4));

    % Determine regression coefficients for mu and sigma for lognormal
    [a, b] = meanAndStdDev(scatter_diagram,0);

    % Fit lognormal distribution to Hs and find parameters mu and sigma
    logn_par = lognfit(metocean(:,3));

    % Transform Hs to lognormal probability
    F_Hs = logncdf(metocean(:,3),logn_par(1),logn_par(2));

    % Transform Tp to lognormal probability conditioned on Hs
    F_Tp = zeros(length(metocean(:,3)),1);
    for ii = 1:length(metocean(:,3))
        % Conditioned mu and sigma values
        mu = a(1) + a(2)*(metocean(ii,3)^a(3));
        sigma = sqrt(b(1) + b(2)*exp(-b(3)*metocean(ii,3)));
        F_Tp(ii) = logncdf(metocean(ii,4),mu,sigma);
    end
    % Inverse standard normal trasnform of Hs and Tp
    [metocean_norm(:,3)] = norminv(F_Hs);
    [metocean_norm(:,4)] = norminv(F_Tp);
else
    % Multivariate BoxCox transform
    lambda_cox = boxcoxn(metocean(:,3:4));
    [metocean_norm(:,3)] = boxcox(lambda_cox(1),metocean(:,3));
```

```
    [metocean_norm(:,4)]= boxcox(lambda_cox(2),metocean(:,4));
end

%% Inverse Seasonal Transformation
[yearMeanFit,yearCovFit,midMonthHours] = seasonal_trans(start_year,...
end_year,metocean_norm,year_length);

% Fit Fourier series to monthly mean and covariance
M_Hs = fit(midMonthHours,yearMeanFit(:,1),'fourier6'); % M Hs
M_Tp = fit(midMonthHours,yearMeanFit(:,2),'fourier6'); % M Tp
S_11 = fit(midMonthHours,yearCovFit(:,1),'fourier6');
S_12 = fit(midMonthHours,yearCovFit(:,2),'fourier6'); % S12=S21
S_22 = fit(midMonthHours,yearCovFit(:,4),'fourier6');

% Reshape simulated values
W_sim = zeros(series_length,2,num_rep);

for ii = 1:num_rep
    W_sim(:,1,ii) = W_sim_raw(:,2*(ii-1)+1);
    W_sim(:,2,ii) = W_sim_raw(:,2*ii);
end

% Inverse seasonal transform
Y_sim_norm = zeros(series_length,2,num_rep);
if std_on == 1
    for ii = 1:num_rep
        for jj = 1:series_length
            M = [M_Hs(jj); M_Tp(jj)];
            S = sqrt([S_11(ii) 0; 0 S_22(ii)]);
            W_inv = [W_sim(jj,1,ii); W_sim(jj,2,ii)];
            Y_sim_norm(jj,:,ii) = M + S*W_inv;
        end
    end
else
    for ii = 1:num_rep
        for jj = 1:series_length
            M = [M_Hs(jj); M_Tp(jj)];
            W_inv = [W_sim(jj,1,ii); W_sim(jj,2,ii)];
            Y_sim_norm(jj,:,ii) = W_inv + M;
        end
    end
end

%% Inverse Rosenblatt transform
Y_sim = zeros(series_length,2,num_rep);

% Standard normal transform
if rosen == 1
    Y_sim_prob = zeros(series_length,2,num_rep);
    % Standard Normal CDF
    for ii = 1:num_rep
        Y_sim_prob(:,1:2,ii) = normcdf(Y_sim_norm(:,1:2,ii));
    end
```

```
    % Inverse lognormal CDF transform
    for ii = 1:num_rep
        Y_sim(:,1,ii) = logninv(Y_sim_prob(:,1,ii),logn_par(1),logn_par(2));
        for jj = 1:series_length
            % mu and sigma vectors for Tp Lognormal transform
            mu = a(1) + a(2)*(Y_sim(jj,1,ii)^a(3));
            sigma = sqrt(b(1) + b(2)*exp(-b(3)*Y_sim(jj,1,ii)));
            Y_sim(jj,2,ii) = logninv(Y_sim_prob(jj,2,ii),mu,sigma);
            if Y_sim(jj,2,ii) == inf
               Y_sim(jj,2,ii) = Y_sim(jj,2,ii-1);
            end
       end
    end
else
    for ii = 1:num_rep
        Y_sim(:,1,ii)=(lambda_cox(1).*Y_sim_norm(:,1,ii)+1).^(1/lambda_cox(1));
        Y_sim(:,2,ii)=(lambda_cox(2).*Y_sim_norm(:,2,ii)+1).^(1/lambda_cox(2));
    end
end

%% Write results to csv
if write == 1
    Y_sim_out = zeros(length(Y_sim),2*num_rep);
    for ii = 1:num_rep
        Y_sim_out(:,2*(ii-1)+1) = Y_sim(:,1,ii);
        Y_sim_out(:,2*ii) = Y_sim(:,2,ii);
    end

    Y_sim_out = [metocean(:,1:2),Y_sim_out];

    csvwrite('VARMA_13_BOXCOX_MS_H.csv',Y_sim_out)
end
```

C.3 Markov Model Scripts

markov_model.m

```
% Markov model
clear all

startYear = 1990; % Start year of time series
endYear = 2015; % Final year of time series, including this year
filename = 'WAM10.txt'; % WAM 10 Metocean file without headers.
numStates = 24; % Set number of states for Hs
numRep = 10;
year_length = 2920;
harsh = 1;

% Load series
if harsh == 1
```

```
    metocean = loadTimeSeries(startYear,endYear,filename);
else
    metocean = csvread(filename);
end

% Load Hs and Tp for the last 30 years from hindcast data
Hs = round(metocean(:,3),1);
Tp = metocean(:,4);

% Length of time series
seriesLength = length(metocean(:,1));

% Number of Replications
numReplications = seriesLength;

% Generate the coupling matrix
C = CouplingMatrix(Hs,Tp);

%% Seasonal Transformation

% Initilize
meanHsMonth = zeros(1,12); stdHsMonth = zeros(1,12);

% Create vector with monthly mean and variance from observed time series
for ii = 1:12
meanHsMonth(ii) = mean(Hs(metocean(:,2) == ii));
stdHsMonth(ii) = std(Hs(metocean(:,2) == ii));
end

% Vector containing the middle day of each month in hours from year start
% Day 1 = 365, such that the period is equal to one year
midMonthHours = transpose([1 15.5 45 74.5 105 135.5 166 196.5 ...
    227.5 258 288.5 319 349.5 365]*(year_length/365));

% Row containing the average mean value for the 1st and 365th day
% Found as the mean of month 1 and month 12
startEndMean = (meanHsMonth(1) + meanHsMonth(end))/2;
startEndSTD = (stdHsMonth(1) + stdHsMonth(end))/2;

% Add the average value row as the first and last element
MeanFit = transpose([startEndMean  meanHsMonth startEndMean]);
stdFit = transpose([startEndMean  stdHsMonth startEndMean]);

% Fit Fourier series to monthly mean values i.e., find periodic M(t)
M = fit(midMonthHours,MeanFit,'fourier6'); % M Hs
S = fit(midMonthHours,stdFit,'fourier5'); % S Hs

% Plot of Fourier approximations for validation of fit
%{
figure(1)
subplot(1,2,1)
plot(M,midMonthHours,MeanFit)
title('M Hs')
subplot(1,2,2)
```

```
plot(S,midMonthHours,stdFit)
title('S Hs')
%}
%% Seasonal transformation

% Initilize
Wobs = zeros(seriesLength,1);

% Perform seasonal transformation W = (Y-M)/S
for ii = 1:seriesLength
    Wobs(ii) = (Hs(ii)-M(ii));%/S(ii);
end

%% ------------------- P Matrix ----------------------------------
% Find upper and lower limits for Hs values
ub = max(Wobs);
lb = min(Wobs);

% Find state ranges
stateRange = (ub-lb) / (numStates);
% State values
stateValues = lb+stateRange:stateRange:ub;
% Preallocation
State = zeros(seriesLength,1);

% Find each data points state
for i = 1:length(Wobs)
    % For each data point
    for j = 1:numStates
        % For each state
        if Wobs(i) <= stateValues(j)
            % Data point is in state j
            State(i) = j;
            % This data point is categorized, so we break and move to the
            % next data point
            break;
        end
    end
end

% Find transitions
transitions = zeros(numStates);

for t = 1:seriesLength-1
    % HsState(t) represents the state and HsState(t+1) represents the state
    % it transitions to
    transitions(State(t),State(t+1)) = transitions(State(t),State(t+1)) + 1;
end

P = transitions;
% Normalize each row in the transition matrix so each row sums to 1
for i = 1:numStates
    P(i,:) = P(i,:) / sum( P(i,:) );
end
```

```
% Check to see if there are any absorbing states
% i.e. P(i,j) == 1 where i=j
absorbstate = zeros(numStates);
for i = 1:numStates
    for j = 1:numStates
        if P(i,j) == 1
            absorbstate(i,j) = absorbstate(i,j) + 1;
        end
    end
end
if sum(sum(absorbstate)) >= 1
    error('Absorbing states');
end

%% Simulation
% Randomly sample start state
state = randi(numStates);

% Preallocation
states = zeros(seriesLength,1);

for kk = 1:numRep
    for i = 1:seriesLength
        % Sample a new random value in range [0,1]
        r = rand();

        for j = 1:numStates
            prob = 0;
            % Accumulate probabilities
            for k = 1:j
                prob = prob + P(state,k);
            end

            if r <= prob
                % New state is found, j
                state = j;

                % Store the state we transition to
                states(i,kk) = j;

                % Break ends the current for loop, and returns to the outer
                % loop, which will sample a new random value and start over
                break;
            end
        end
    end
end

% Transform states to normalized values
WSim = zeros(seriesLength,numRep);
for jj = 1:numRep
    for ii = 1:seriesLength
        WSim(ii,jj) = stateValues(states(ii,jj));
```

```
    end
end

%% ------------------------- Inverse Transform --------------------------
% Preallocation
HsSim = zeros(seriesLength,numRep);

% Apply inverse
for jj = 1:numRep
    for ii = 1:seriesLength
        HsSim(ii,jj) = M(ii) + WSim(ii,jj);%*S(ii);
        if HsSim(ii,jj) < min(Hs)
            HsSim(ii,jj) = min(Hs);
        end
    end
end

%% ------------------------- Coupling ----------------------------------
Tp_unique = unique(Tp); Hs_unique = unique(Hs);
n_unique_Hs = length(Hs_unique); n_unique_Tp = length(Tp_unique);
simTpSpread = zeros(seriesLength,numRep);

for kk = 1:numRep
% Preallocation
Hs_state = zeros(seriesLength,1);

% Find closest unique Hs state
for ii = 1:seriesLength
    for jj = 1: n_unique_Hs
        if HsSim(ii,kk) <= Hs_unique(jj)
            Hs_state(ii) = jj;
            % Break inner loop when state is found
            break;
        end
    end
end

% Can try to enter state max + 1
Hs_state(Hs_state == 0) = max(Hs_state);

% Preallocate Tp coupled vector
Tp_coupled = zeros(1,seriesLength);

for jj = 1:seriesLength

r = rand; % Draw random number between 0 and 1
prob = 0; % Reset probabilty for each replication

    for ii = 1:n_unique_Tp
        % Accumulate probabilities
        prob = prob + C(Hs_state(jj),ii);

        % Find Tp values
        if r <= prob
```

```
            Tp_coupled(jj) = Tp_unique(ii);
            % Break inner loop when state is found
            break;
        end
    end
end

%% Spread T_p Values

if harsh == 1
    % Simulated values
    % Preallocation
    Tp_coupled_spread = zeros(1,length(Tp_coupled));

    % Apply spreading function to simulated Tp
    for ii = 1:length(Tp_coupled)
        n = round(1+log(Tp_coupled(ii)/3.244)/0.09525);
        Tp_coupled_spread(ii) = 3.244*exp(0.09525*(n-0.5-rand));
    end

    simTpSpread(:,kk) = Tp_coupled_spread;

else
    metocean(:,4) = Tp_coupled;
    simTpSpread(:,kk) = spread_calm_peak_period(metocean);
end
end

if harsh == 1
    % Observed values spread_calm_peak_period(metocean)
    % Preallocation
    Tp_spread = zeros(length(Tp),1);

    % Apply spreading function to observed Tp
    for ii = 1:length(Tp)
        n = round(1+log(Tp(ii)/3.244)/0.09525);
        Tp_spread(ii,1) = 3.244*exp(0.09525*(n-0.5-rand));
    end
else
    metocean(:,4) = Tp;
    Tp_spread = spread_calm_peak_period(metocean);
end

%% Write Series Ysim to csv

Ysim = zeros(seriesLength,2,numRep);

for ii = 1:numRep
Ysim(:,1,ii) = HsSim(:,ii);
Ysim(:,2,ii) = simTpSpread(:,ii);
end

indices = [metocean(:,1) metocean(:,2)];
for ii = 1:numRep
```

```
    if ii == 1
        sims = [Ysim(:,1,ii) Ysim(:,2,ii)];
    else
        sims = [sims Ysim(:,1,ii) Ysim(:,2,ii)];
    end
end
out = [indices sims];
csvwrite('Markov_M_H.csv',out)
```

CouplingMatrix.m

```
function C_prob = CouplingMatrix(Hs,Tp)

% Create vector of unique observed values
Tp_unique = unique(Tp);
Hs_unique = unique(Hs);

% Lengths
n_unique_Hs = length(Hs_unique);
n_unique_Tp = length(Tp_unique);

% Preallocate
C_events = zeros(n_unique_Hs,n_unique_Tp);
C_prob = zeros(n_unique_Hs,n_unique_Tp);

% --------------------- Count Occurences ---------------------------------
for kk = 1:n_unique_Hs
    % Find all indexes in metocean data for each unique Hs
    index = find(Hs == Hs_unique(kk));

    % Count number of occurences of each unique Tp for each  unique Hs
    for ii = 1:length(index)
        for jj = 1:n_unique_Tp
            if Tp(index(ii)) == Tp_unique(jj)
                C_events(kk,jj) = C_events(kk,jj) + 1;
                break;
            end
        end
    end
end

%Normalize to probabilty matrix
for ii = 1:n_unique_Hs
    C_prob(ii,:) = C_events(ii,:)/sum(C_events(ii,:));
end
end
```

C.4 VAR Model Script

VAR_model.m

```
% VAR estimation and simulation of residuals. Includes back-transform of
% residuals

clear all

% General parameters
start_year = 1990; % Start year of time series
end_year = 2015; % Final year of time series, including this year
year_length = 2920; % 1460 Number of observations in a year
filename = 'WAM10.txt'; % WAM 10 Metocean file without headers.
harsh = 1; % 0 for calm, 1 for
num_rep = 10; % Number of replications
write_to_csv = 0; % write residuals
write_filename = 'VAR_XX_MS_C.csv';

% VAR paramaters
np = 10; % Number of p lags to be evaluated. Integer and larger than zero
burnIn = 400; % Number of time steps for burn-in. 200 is normal

% Load Hs, Tp, year indices and month indices from start year to and
% including end year
if harsh == 1
    metocean = loadTimeSeries(start_year,end_year,filename);
else
    metocean = csvread(filename);
end

% Length of time series
series_length = length(metocean(:,1));

%% Uniformly distribute Tp
if harsh == 1
    % Spread Tp with uniform distribution
    Tp_spread = zeros(length(metocean),1);
    for ii = 1:length(metocean(:,4))
        n = round(1+log(metocean(ii,4)/3.244)/0.09525);
        metocean(ii,4) = 3.244*exp(0.09525*(n-0.5-rand));
    end
else
    metocean(:,4) = spread_calm_peak_period(metocean);
end

%% Transformation to normality
% Allocate year and month indices
metocean_norm(:,1) = metocean(:,1);
metocean_norm(:,2) = metocean(:,2);

% Rosenblatt transform
```

```
%Creating scatter diagram of
scatter_diagram = scatterDiagram(metocean(:,3),metocean(:,4));

% Determine regression coefficients for mu and sigma for lognormal
[a, b] = meanAndStdDev(scatter_diagram,0);

% Fit lognormal distribution to Hs and find parameters mu and sigma
logn_par = lognfit(metocean(:,3));

% Transform Hs to lognormal probability
F_Hs = logncdf(metocean(:,3),logn_par(1),logn_par(2));

% Transform Tp to lognormal probability conditioned on Hs
F_Tp = zeros(length(metocean(:,3)),1);
for ii = 1:length(metocean(:,3))
    % Conditioned mu and sigma values
    mu = a(1) + a(2)*(metocean(ii,3)^a(3));
    sigma = sqrt(b(1) + b(2)*exp(-b(3)*metocean(ii,3)));
    F_Tp(ii) = logncdf(metocean(ii,4),mu,sigma);
end

% Inverse standard normal trasnform of Hs and Tp
[metocean_norm(:,3)] = norminv(F_Hs);
[metocean_norm(:,4)] = norminv(F_Tp);

%% Seasonal Transformation
[yearMeanFit,yearCovFit,midMonthHours] = seasonal_trans(start_year,...
end_year,metocean_norm,year_length);

% Fit Fourier series to monthly mean and covariance
M_Hs = fit(midMonthHours,yearMeanFit(:,1),'fourier6'); % M Hs
M_Tp = fit(midMonthHours,yearMeanFit(:,2),'fourier6'); % M Tp
S_11 = fit(midMonthHours,yearCovFit(:,1),'fourier6');
S_12 = fit(midMonthHours,yearCovFit(:,2),'fourier6'); % S12=S21
S_22 = fit(midMonthHours,yearCovFit(:,4),'fourier6');

%% Generate Residuals
% Pre-allocate
W = zeros(series_length,2);

% Perform seasonal transformation W = (Y-M)/S
for ii = 1:series_length
    Y = [metocean_norm(ii,3); metocean_norm(ii,4)];
    M = [M_Hs(ii); M_Tp(ii)];
    S = sqrt([S_11(ii) 0; 0 S_22(ii)]);
    W(ii,:)=(Y-M);
    %W(ii,:)=S\(Y-M); % x = A\b --- Ax = b
end

%% Find optimal VAR model lags for residuals W with AIC
% Initilize
LOGL = zeros(np,1);

% Fit VAR(p) models
```

```
for p = 1:np
        mdl = varm(2,p);
        [fit,~,logL] = estimate(mdl,[W(:,1) W(:,2)]);
        % Store the loglikelihood objective function and number of
        % coefficients for each fitted model
        LOGL(p) = logL;
end

% Column vector with number of terms for each VAR(p)
P(:,1) = 2+4*[1:np];

% Calculate the AIC/BIC for each fitted VAR model
[AIC,BIC] = aicbic(LOGL,P,series_length);

% Optimal number of lags according to AIC.
[minNumAIC,pLagsAIC] = min(AIC);

%% Simulate Residuals W
% Re-fit optimal VAR model to W residuals
mdlW = varm(2,pLagsAIC);
estMdlW = estimate(mdlW,[W(:,1) W(:,2)]);

% Simulate series
WsimBurnIn = simulate(estMdlW,burnIn+series_length,'NumPaths',num_rep);
W_sim = WsimBurnIn(burnIn+1:end,:,:);

%% Inverse Seasonal Transform Y = M + S*W
Y_sim_norm = zeros(series_length,2,num_rep);
for ii = 1:num_rep
    for jj = 1:series_length
        M = [M_Hs(jj); M_Tp(jj)];
        S = sqrt([S_11(ii) 0; 0 S_22(ii)]);
        W_inv = [W_sim(jj,1,ii); W_sim(jj,2,ii)];
        %Y_sim_norm(jj,:,ii) = W_inv + M;
        Y_sim_norm(jj,:,ii) = M + S*W_inv;
    end
end

%% Inverse Rosenblatt transform
% Standard normal transform
Y_sim_prob = zeros(series_length,2,num_rep);
for ii = 1:num_rep
    Y_sim_prob(:,1:2,ii) = normcdf(Y_sim_norm(:,1:2,ii));
end

% Inverse lognormal CDF transform
Y_sim = zeros(series_length,2,num_rep);
for ii = 1:num_rep
    Y_sim(:,1,ii) = logninv(Y_sim_prob(:,1,ii),logn_par(1),logn_par(2));
    for jj = 1:series_length
    % mu and sigma vectors for Tp Lognormal transform
    mu = a(1) + a(2)*(Y_sim(jj,1,ii)^a(3));
    sigma = sqrt(b(1) + b(2)*exp(-b(3)*Y_sim(jj,1,ii)));
    Y_sim(jj,2,ii) = logninv(Y_sim_prob(jj,2,ii),mu,sigma);
```

```
    end
end

%% Write simulated series to .csv
if write_to_csv == 1
    indices = [metocean(:,1) metocean(:,2)];
    for ii = 1:num_rep
        if ii == 1
            y_sim_out = [Y_sim(:,1,ii) Y_sim(:,2,ii)];
        else
            y_sim_out = [y_sim_out Y_sim(:,1,ii) Y_sim(:,2,ii)];
        end
    end
    out = [indices y_sim_out];
    csvwrite(write_filename,out)
end
```

C.5 VARMA Model Script

Note that this script is written in R.

VARMA_model.r

```
# Simulate residuals using VARMA model

# Set directory
setwd("C:/Users/olejl/OneDrive/Dokumenter/MATLAB/MasterThesis")

# Open time series library
library(MTS)

# Load residuals from Matlab
mydata = read.table("C:/residuals_BOXCOX_MS_H.csv",header=FALSE,sep=",")

# Select Hs and Tp columns
mydata = mydata[,3:4]

# Calculate cross-corrolation P values
P=Eccm(mydata,maxp=4,maxq=4)

# P values matrix
pValues = P$pEccm

# Find the smallest P value larger than zero
pValueMin = which(pValues == min(pValues[pValues > 0.05]),arr.ind=T)

# Optimal lags by P value
pLags = pValueMin[1]-1
qLags = pValueMin[2]-1
```

```
# Lag vectors
ARlags = c(1:pLags)
MAlags = c(1:qLags)

# Fit VARMA model to residuals W
mdl=VARMA(mydata,p=pLags,q=qLags,include.mean=FALSE)

# VARMA parameters
AIC = mdl$aic
phi = mdl$Phi
theta = mdl$Theta
sig = mdl$Sigma

series_length = length(mydata)

# Simulate VARMA
m=VARMAsim(series_length,arlags=ARlags,malags=MAlags,
phi=phi,theta=theta,sigma=sig)

# Simulated residual series
Wsim=m$series

# Write simulation results to .csv
write.table(Wsim,file="Residuals.csv",row.names=FALSE,na="",
col.names=FALSE,sep=",")
```

C.6 SAR-LSTM Network Model Scripts

Note that these scripts are written in Python.

grid_search_LSTM.py

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
from keras import regularizers

################## Functions ###################

# Scale features
def scale_features(metocean_data):
    # Transform dataframe to floats
    raw_values = metocean_data.values
    #raw_values = raw_values[0:series_length,:]

    # Scale features
```

```
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled = scaler.fit_transform(raw_values)
    return scaler,scaled

# convert series to supervised learning
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg

# split supervised problem into train and test
def supervised_to_train_test(reframed,batch_size,window):
    # Transform to float
    values = reframed.values
    # Split into training and test set
    train_size = int(len(reframed) * 0.80)
    train, test = values[0:train_size], values[train_size:len(reframed)]
    # split into input and outputs
    train_X, train_Y = train[:, :-num_features], train[:, -num_features:]
    test_X, test_Y = test[:, :-num_features], test[:, -num_features:]
    # reshape input to be 3D [samples, timesteps, features]
    train_X = train_X.reshape((train_X.shape[0], window, num_features))
    test_X = test_X.reshape((test_X.shape[0], window, num_features))
    # adjust length of training and test sets to be a factor of the batch size
    num_batches_train = int(len(train_X)/batch_size)
    num_batches_test = int(len(test_X)/batch_size)
    train_X = train_X[0:(num_batches_train*batch_size),:,:]
    train_Y = train_Y[0:(num_batches_train*batch_size),:]
    test_X = test_X[0:(num_batches_test*batch_size),:,:]
    test_Y = test_Y[0:(num_batches_test*batch_size),:]
    # Drop months as feature
    train_Y = train_Y[:,1:]
    test_Y = test_Y[:,1:]
    return train_X, train_Y, test_X, test_Y

# Network Architecture
```

```
def build_LSTM(neurons,stack_depth,batch_size):
    model = Sequential()
    if stack_depth > 1:
        for i in range(stack_depth):
            model.add(LSTM(neurons,batch_input_shape=(batch_size,
                train_X.shape[1], train_X.shape[2]),
                stateful=True,
                return_sequences=True,
                kernel_regularizer=regularizers.l2(lambda_reg)))
            model.add(Dropout(0.50))
        model.add(LSTM(neurons,batch_input_shape=(batch_size,
                    train_X.shape[1], train_X.shape[2]),
                    stateful=True,
                    kernel_regularizer=regularizers.l2(lambda_reg)))
    else:
        model.add(LSTM(neurons,batch_input_shape=(batch_size,
                    train_X.shape[1], train_X.shape[2]),
                    stateful=True,
                    kernel_regularizer=regularizers.l2(lambda_reg)))
    model.add(Dropout(0.50))
    model.add(Dense(2))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# Train model and return loss history
def train_LSTM(num_epochs,batch_size):
    run_history = np.zeros((num_epochs,2))
    for i in range(num_epochs):
        History = model.fit(train_X, train_Y, epochs=1, batch_size=batch_size,
                    validation_data=(test_X,test_Y), verbose=2, shuffle=False)
        model.reset_states()
        print('True Epoch:', (i+1))
        loss_dict = History.history
        run_history[i,0] = np.array(loss_dict['loss'])
        run_history[i,1] = np.array(loss_dict['val_loss'])
        run_history = np.array(run_history)
    return run_history

################## End Functions ##################

# Load the dataset
metocean_data = pd.read_csv('W:/Residuals/residuals_rosen_M_C.csv',
                                                    header=None,usecols=[1,2,3],
                names=['month','Hs','Tp'], engine='python')

# Non-variable parameters
num_features = 3
num_epochs = 60
batch_size = 64

# Scale features
scaler,scaled = scale_features(metocean_data)

# Hyper-parameter values
```

```
window = [8,16,32]
neurons = [32,64,128]
stack_depth = [1,2,3]
lambda_reg = 0.00001

# Counter for iteration number
counter = 1

for i in range(len(window)):
    for j in range(len(neurons)):
        for k in range(len(stack_depth)):
            # Frame as supervised learning
            reframed = series_to_supervised(scaled, window[i], 1)

            # Split into train and test sets
                        # Adjust length to be a factor of batch size
            train_X,train_Y,test_X,test_Y = supervised_to_train_test(reframed,


            # Build Network Architecture
            model = build_LSTM(neurons[j],stack_depth[k],batch_size)

            # Train Network
            run_history = train_LSTM(num_epochs,batch_size)

            # Save loss history and model parameters
            # Parameter column
            run_parameters = np.array([[window[i],neurons[j],stack_depth[k],

            run_parameters = np.concatenate((run_parameters,run_parameters)).T

            # Append loss history to parameters
            run_history = np.append(run_parameters,run_history,axis=0)
            #run_history = pd.DataFrame(run_history)
            # Add loss and parameters to complete history
            if counter == 1:
                full_history = run_history

            else:
                full_history = np.column_stack((full_history,run_history))
                # Write to csv after each iteration
                history_df = pd.DataFrame(full_history)
                history_df.to_csv('W:/output_pyy.csv',
                                  header=False, index=False)
            counter = counter + 1
```

train_single_LSTM.py

```
# Train single network model

import pandas as pd
```

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout
from keras import regularizers
################################## FUNCTIONS ###################################

# Scale features
def scale_features(metocean_data):
    # Transform dataframe to floats
    raw_values = metocean_data.values
    #raw_values = raw_values[0:series_length,:]

    # Scale features
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled = scaler.fit_transform(raw_values)
    return scaler,scaled

# convert series to supervised learning
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg

# split supervised problem into train and test
def supervised_to_train_test(reframed,batch_size,window):
    # Transform to float
    values = reframed.values
    # Split into training and test set
    train_size = int(len(reframed) * 0.80)
    train, test = values[0:train_size], values[train_size:len(reframed)]
    # split into input and outputs
    train_X, train_Y = train[:, :-num_features], train[:, -num_features:]
    test_X, test_Y = test[:, :-num_features], test[:, -num_features:]
    # reshape input to be 3D [samples, timesteps, features]
    train_X = train_X.reshape((train_X.shape[0], window, num_features))
```

```
    test_X = test_X.reshape((test_X.shape[0], window, num_features))
    # adjust length of training and test sets to be a factor of the batch size
    num_batches_train = int(len(train_X)/batch_size)
    num_batches_test = int(len(test_X)/batch_size)
    train_X = train_X[0:(num_batches_train*batch_size),:,:]
    train_Y = train_Y[0:(num_batches_train*batch_size),:]
    test_X = test_X[0:(num_batches_test*batch_size),:,:]
    test_Y = test_Y[0:(num_batches_test*batch_size),:]
    # Drop months as feature
    train_Y = train_Y[:,1:]
    test_Y = test_Y[:,1:]
    return train_X, train_Y, test_X, test_Y

# Network Architecture
def build_LSTM(neurons,stack_depth,batch_size):
    model = Sequential()
    for i in range(stack_depth):
        model.add(LSTM(neurons,batch_input_shape=(batch_size,
            train_X.shape[1], train_X.shape[2]),
            stateful=True,
            return_sequences=True,
            kernel_regularizer=regularizers.l2(lambda_reg)))
        model.add(Dropout(0.50))
    model.add(LSTM(neurons,batch_input_shape=(batch_size,
                train_X.shape[1], train_X.shape[2]),
                stateful=True,
                kernel_regularizer=regularizers.l2(lambda_reg)))
    model.add(Dropout(0.50))
    model.add(Dense(2))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# Train model and return loss history
def train_LSTM(num_epochs,batch_size):
    run_history = np.zeros((num_epochs,2))
    best_val_loss = 10;
    for i in range(num_epochs):
        History = model.fit(train_X, train_Y, epochs=1, batch_size=batch_size,
                    validation_data=(test_X,test_Y), verbose=2, shuffle=False)
        model.reset_states()
        print('True Epoch:', (i+1))
        loss_dict = History.history
        run_history[i,0] = np.array(loss_dict['loss'])
        run_history[i,1] = np.array(loss_dict['val_loss'])
        run_history = np.array(run_history)
        if run_history[i,1] < best_val_loss:
                model_best = model
    return run_history,model_best
##################################################################

# Load the dataset
metocean_data = pd.read_csv('W:/Residuals/residuals_rosen_M_C.csv',header=None,
                                                    usecols=[1,2,3],
                        names=['month','Hs','Tp'], engine='python')
```

```
# Non-variable parameters
num_features = 3
num_epochs = 500
batch_size = 64

# Scale features
scaler,scaled = scale_features(metocean_data)

# Hyper-parameter values
window = 16
neurons = 128
stack_depth = 2
lambda_reg = 0

# Frame as supervised learning
reframed = series_to_supervised(scaled, window, 1)

# Split into train and test sets and adjust length to be a factor of batch size
train_X, train_Y, test_X, test_Y = supervised_to_train_test(reframed,
                                   batch_size,window)

# Build Network Architectur
model = build_LSTM(neurons,stack_depth,batch_size)

# Train Network
run_history,model_best = train_LSTM(num_epochs,batch_size)

# Save loss history and model parameters
# Parameter column
run_parameters = np.array([[window,neurons,stack_depth,lambda_reg]])
run_parameters = np.concatenate((run_parameters,run_parameters)).T

# Append loss history to parameters
run_history = np.append(run_parameters,run_history,axis=0)

# Write loss to csv
history_df = pd.DataFrame(run_history)
history_df.to_csv('W:/Models/loss_LSTM_model_1_C.csv',
                  header=False, index=False)

# Change batch size of model
if batch_size != 1:
    # re-define model with batch size of 1
    new_model = build_LSTM(neurons,stack_depth,1)

    # copy weights from old model and apply to new model
    old_weights = model_best.get_weights()
    new_model.set_weights(old_weights)

# Save new model to current directory
new_model.save('W:/Models/LSTM_model_1_C.h5')
```

simulate_LSTM.py

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from keras.models import load_model
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import norm
from sklearn.metrics import mean_squared_error
import math

################################## FUNCTIONS ###################################
# convert series to supervised learning
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg

# split supervised problem into train and test
def supervised_to_train_test(reframed,batch_size,window):
    # Transform to float
    values = reframed.values
    # Split into training and test set
    train_size = int(len(reframed) * 0.80)
    train, test = values[0:train_size], values[train_size:len(reframed)]
    # split into input and outputs
    train_X, train_Y = train[:, :-num_features], train[:, -num_features:]
    test_X, test_Y = test[:, :-num_features], test[:, -num_features:]
    # reshape input to be 3D [samples, timesteps, features]
    train_X = train_X.reshape((train_X.shape[0], window, num_features))
    test_X = test_X.reshape((test_X.shape[0], window, num_features))
    # adjust length of training and test sets to be a factor of the batch size
    num_batches_train = int(len(train_X)/batch_size)
```

```
    num_batches_test = int(len(test_X)/batch_size)
    train_X = train_X[0:(num_batches_train*batch_size),:,:]
    train_Y = train_Y[0:(num_batches_train*batch_size),:]
    test_X = test_X[0:(num_batches_test*batch_size),:,:]
    test_Y = test_Y[0:(num_batches_test*batch_size),:]
    # Drop months as feature
    train_Y = train_Y[:,1:]
    test_Y = test_Y[:,1:]
    return train_X, train_Y, test_X, test_Y

def simulate_LSTM(num_rep):
    # Initilize starting window for simulation
    trainPredict = [train_X[0,:,:]]
    # Pre-allocate prediction output
    predictions = np.zeros((num_rep,2))
    # Simulate one new prediction for the length of the simulation
    for i in range(num_rep):
        prediction = model.predict(np.array([trainPredict[-1]]),batch_size=1)
        # inverse transform to residual scale
        prediction = scaler_pred.inverse_transform(prediction)
        # Add white noise to inverse transformed predictions
        prediction[0,0] = prediction[0,0] + np.random.normal(0,sigma_obs[0],1)
        prediction[0,1] = prediction[0,1] + np.random.normal(0,sigma_obs[1],1)
        # Transform back to prediction scale
        prediction = scaler_pred.transform(prediction)
        # Store all predictions
        predictions[i] = prediction
        # Append last value for month to shape for next prediction
        prediction = np.insert(prediction,0,[trainPredict[-1][-1][0]])
        prediction.flatten()
        # Append predictions as the value of the previous time step
        trainPredict.append(np.vstack([trainPredict[-1][1:],prediction]))
    # Inverse transform all predictions to residual scale
    predictions = scaler_pred.inverse_transform(predictions)
    # transform to dataframe
    df_predictions = pd.DataFrame(predictions,columns =['Hs','Tp'])
    return df_predictions

################################### END OF FUNC ##############################


# Load stationary residuals, month and year
df_observed = pd.read_csv('residuals_rosen_M_C.csv',header=None,
            usecols=[0,1,2,3],names=['year','month','Hs','Tp'], engine='python')

# PARAMETERS
num_features = 3
num_rep = 37960 # Number of replications

# HYPER-PARAMETERS
look_back = 16
neurons = 128
batch_size = 64
stack_depth = 2
```

```
# Select years for training and testing
df_observed = df_observed[['month','Hs','Tp']]

# Mean and STD of observations
sigma_obs = [np.std(df_observed['Hs']), np.std(df_observed['Tp'])]
mu_obs = [np.mean(df_observed['Hs']), np.mean(df_observed['Tp'])]

# Scale all features
scaler_all = MinMaxScaler(feature_range=(0, 1))
scaled = scaler_all.fit_transform(df_observed.values)

# New scaler to only apply to Hs and Tp
scaler_pred = MinMaxScaler(feature_range=(0, 1))
scaler_pred.fit(df_observed[['Hs','Tp']].values)

# Frame as supervised learning
reframed = series_to_supervised(scaled, look_back, 1)

# Split into train and test sets and adjust length to be a factor of batch size
train_X, train_Y, test_X, test_Y = supervised_to_train_test(reframed,
                                                            64,look_back)

# Load deep learning model
model = load_model('LSTM_model_1_H.h5')

# Re-define batch size
batch_size = 1
# ---------------------- Forecast ------------------------------------------

# make predictions
test_predict = model.predict(test_X, batch_size=batch_size)

# Inverse scale
test_predict = scaler_pred.inverse_transform(test_predict)
test_Y = scaler_pred.inverse_transform(test_Y)

# Prediction test score
RMSE_Hs = math.sqrt(mean_squared_error(test_Y[:,0], test_predict[:,0]))
print('Test Score: %.2f RMSE' % (RMSE_Hs))

# ---------------------- Simulate ------------------------------------------


# Simulate
for i in range(10):
    df_predictions = simulate_LSTM(num_rep,scale_val)
    if i == 0:
        synthetic_series = df_predictions
    else:
        synthetic_series = pd.concat([synthetic_series, df_predictions],axis=1)


synthetic_series.to_csv('C:/Users/olejl/LSTM/residuals_LSTM_M_C.csv',
```

```
                    header=False, index=False)
```

C.7 Validation Scripts

validate_single.m

```
% Validate a single synthetic series against observed
clear all

% General parameters
harsh = 1; % 0 for calm, 1 for harsh
num_rep = 10;
lstm = 0;

% plot parameters
m_type = 'LSTM$_m$';
line_width = 1.5;
num_lags_auto = 100; % Number of lags in auto/crosscorrelation
plot_pers = 0;

% Files
observed_filename = 'WAM10.txt'; % WAM 10 Metocean file without headers.
synthetic_filename = 'VARMA_13_BOXCOX_MS_H.csv';

% LOAD PERSISTENCE TABLES
if plot_pers == 1
    obs_persistence = xlsread('pers_table_harsh.xlsx');
    obs_above = obs_persistence(2:61,1:21);
    obs_below = obs_persistence(63:122,1:21);
    syn_persistence = xlsread('pers_table_LSTM_M_H.xlsx');
    syn_above = syn_persistence(2:61,1:21);
    syn_below = syn_persistence(63:122,1:21);
end

% Read observed series
if harsh == 1
    step_length = 3;
    start_year = 1990; % Start year of time series
    end_year = 2015; % Final year of time series, including this year
    year_length = 2920; % 1460 Number of observations in a year
    metocean = loadTimeSeries(start_year,end_year,observed_filename);
    scatter_max_x = 22;
    above_values = [.001 .01 .1 .2 .4 .6 .7]; % [.001 .01 .1 .2 .4 .6 .7];
    below_values = [.01 .1 .2 .4 .6 .7 .8]; %[.01 .1 .2 .4 .6 .7 .8];
else
    metocean = csvread(observed_filename);
    step_length = 6;
    year_length = 1460;
    scatter_max_x = 16;
    above_values = [.001 .01 .05 .1 .2 .6 .7];
```

```
    below_values = [.15 .3 .45 .65 .8 .9 .9];
end

%% Set Latex as interpreter and font size
set(0,'defaultTextInterpreter','latex')
set(0, 'defaultAxesTickLabelInterpreter','latex');
set(0, 'defaultLegendInterpreter','latex');

set(0,'DefaultAxesFontSize', 16) % 12 for 0.8, 16 for 0.65
set(0,'DefaultTextFontSize', 16)
set(0,'DefaultLegendFontSize', 16)

%% Read
% Read synthetic series
syn_series = csvread(synthetic_filename);

% Remove month and year columns
syn_series = syn_series(:,3:end);

% Reshape synthetic series
Y_sim = zeros(length(syn_series),2,num_rep);
for ii = 1:num_rep
    Y_sim(:,1,ii) = syn_series(:,2*(ii-1)+1);
    Y_sim(:,2,ii) = syn_series(:,2*ii);
end


%% Uniformly distribute Tp
if harsh == 1
    % Spread Tp with uniform distribution
    for ii = 1:length(metocean(:,4))
        n = round(1+log(metocean(ii,4)/3.244)/0.09525);
        metocean(ii,4) = 3.244*exp(0.09525*(n-0.5-rand));
    end
else
    metocean(:,4) = spread_calm_peak_period(metocean);
end

Hs_obs = metocean(:,3);
Tp_obs = metocean(:,4);

%% Overall and Monthly Mean and Covaraince
% Over-all mean and covariance
overall_mean_cov = over_all_mean_variance(metocean(:,3),...
                    metocean(:,4),Y_sim,num_rep);

% Monthly Mean and Covariance
[obs_cov,sim_cov,obs_month_mean,sim_month_mean]=monthly_mean_covariance(...
                            metocean,num_rep,Y_sim);

% Plot monthly mean and covariance
figure(1)
clf
plot(1:12,obs_month_mean(1,:),'LineWidth',line_width)
```

```
hold on
plot(1:12,sim_month_mean(1,:),'LineWidth',line_width,'LineStyle','--')
plot(1:12,obs_month_mean(2,:),'LineWidth',line_width)
plot(1:12,sim_month_mean(2,:),'LineWidth',line_width,'LineStyle','--')
legend({'Observed $H_s$',[m_type ' $H_s$'],'Observed $T_p$',[m_type ' $T_p$']})
% title([m_type ' Comparison Plot of Monthly Mean for Series ' s_type])
xlabel('Month','Color','k')
ylabel('Mean','Color','k')
xticks([1 3 6 9 12])
xlim([1,12])
if harsh == 1
    ylim([0,14])
else
    ylim([0,14]) % 10.5 for NOT LSTM
end
grid on


figure(2)
clf
plot(1:12,obs_cov(1,:),'LineWidth',line_width)
hold on
plot(1:12,sim_cov(1,:),'LineWidth',line_width,'LineStyle','--')
plot(1:12,obs_cov(2,:),'LineWidth',line_width)
plot(1:12,sim_cov(2,:),'LineWidth',line_width,'LineStyle','--')
plot(1:12,obs_cov(3,:),'LineWidth',line_width)
plot(1:12,sim_cov(3,:),'LineWidth',line_width,'LineStyle','--')
legend('Observed $\Sigma_{11}$',[m_type ' $\Sigma_{11}$'],...
    'Observed $\Sigma_{12}$',[m_type ' $\Sigma_{12}$'],...
    'Observed $\Sigma_{22}$',[m_type ' $\Sigma_{22}$']);
%title('Comparison Plot of Monthly Covariance')
xlabel('Month','Color','k')
ylabel('Covariance','Color','k')
xticks([1 3 6 9 12])
xlim([1,12])
if harsh == 1
    ylim([0,24]) % 8 NOT LSTM
else
    ylim([0,30]) % 5.5 NOT LSTM
end
grid on

%% Autocorrelation and Crosscorrelation

% Hs Autocorrelation
[acf_Hs_sim,lags_auto] = autocorr(Y_sim(:,1,1),num_lags_auto);
acf_Hs_obs = autocorr(metocean(:,3),num_lags_auto);

% Tp Autocorrelation
acf_Tp_sim = autocorr(Y_sim(:,2,1),num_lags_auto);
acf_Tp_obs = autocorr(metocean(:,4),num_lags_auto);

figure(3)
clf
```

```
subplot(2,1,1)
plot(lags_auto,acf_Hs_obs,'LineWidth',line_width)
hold on
plot(lags_auto,acf_Hs_sim,'LineWidth',line_width,'LineStyle','--')
%title('$H_s$ Autocorrelation')
xlabel('Lag','Color','k')
ylabel('ACF $H_s$','Color','k')
legend('Observed',m_type)
grid on
subplot(2,1,2)
plot(lags_auto,acf_Tp_obs,'LineWidth',line_width)
hold on
plot(lags_auto,acf_Tp_sim,'LineWidth',line_width,'LineStyle','--')
%title('$T_p$ Autocorrelation')
%legend('Observed',m_type)
xlabel('Lag','Color','k')
ylabel('ACF $T_p$','Color','k')
grid on

% Cross Correlation
[xcf_sim,lags_cross] = crosscorr(Y_sim(:,1,1),Y_sim(:,2,1));
xcf_obs = crosscorr(metocean(:,3),metocean(:,4));

figure(4)
clf
plot(lags_cross,xcf_obs,'LineWidth',line_width)
hold on
plot(lags_cross,xcf_sim,'LineWidth',line_width,'LineStyle','--')
xlabel('Lag','Color','k')
ylabel('Crosscorrelation Function','Color','k')
legend('Observed',m_type)
grid on
%title('Crosscorrelation')


%% Scatter plot
series_length = length(metocean);

% Preallocation
group1 = cell(series_length,1);
group2 = cell(series_length,1);

% Create one group for model and observed
for ii = 1:series_length
    group1{ii} = 'Observed';
    group2{ii} = m_type;
end

% Vectors for scatterhist
type = cat(1,group1,group2);
HsPlot = [metocean(:,3); Y_sim(:,1,1)];
TpPlot = [metocean(:,4); Y_sim(:,2,1)];

%[0 2 4 6 8 10 12 14 16 18 20 22]
```

```
figure(5)
clf
scatterhist(TpPlot, HsPlot,'Group',type, 'MarkerSize', 2,'kernel','on',...
    'Color','br','LineWidth',[line_width,line_width],'Location','SouthWest')
xlim([0 28]) % scatter_max_x
ylim([0 36])
xticks(0:4:28) % scatter_max_x
%xticklabels({'2','4','6','8','10','12','14','16','18','20','22',})
yticks(0:6:36)
ylabel('$H_s$','Color','k')
xlabel('$T_p$','Color','k')
%[lgd,b] = legend('Observed',m_type);
%lgd.Location = 'none';
%set(findobj(b,'-property','MarkerSize'),'MarkerSize',10)
grid on

%% Normalized Persistence Tables

if plot_pers == 1

        % Total simulated time
    total_time = series_length * step_length;

    % Size of persistence tables
    [nRow, nCol] = size(obs_above);

    % Initilize
    hindAboveTime = zeros(nRow,nCol); hindBelowTime = zeros(nRow,nCol);
    simAboveTime = zeros(nRow,nCol); simBelowTime = zeros(nRow,nCol);

    % Normalize and transform to real time
    for ii = 1:nRow
        hindAboveTime(ii,:) = 6*ii*obs_above(ii,:)/total_time;
        hindBelowTime(ii,:) = 6*ii*obs_below(ii,:)/total_time;
        simAboveTime(ii,:) = 6*ii*syn_above(ii,:)/total_time;
        simBelowTime(ii,:) = 6*ii*syn_below(ii,:)/total_time;
    end
    % Low Values Normalized Persistence ABOVE
    figure(6)
    clf
    v = [above_values(1),above_values(1)];
    [C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    hold on
    [C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
    clabel(C,h)

    v = [above_values(2),above_values(2)];
    [C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    hold on
    [C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
    clabel(C,h)
```

```
v = [above_values(3),above_values(3)];
[C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
clabel(C,h)
[C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
clabel(C,h)

v = [above_values(4),above_values(4)];
[C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
clabel(C,h)
hold on
[C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
clabel(C,h)

v = [above_values(5),above_values(5)];
[C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
clabel(C,h)
hold on
[C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
clabel(C,h)

v = [above_values(6),above_values(6)];
[C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
clabel(C,h)
hold on
[C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
clabel(C,h)

v = [above_values(7),above_values(7)];
[C,h] = contour(flipud(hindAboveTime),v,'b','LineWidth',line_width);
clabel(C,h)
[C,h] = contour(flipud(simAboveTime),v,'--r','LineWidth',line_width);
clabel(C,h)
yticks([1 10 20 30 40 50 60])
yticklabels({'360','300','240','180','120','60','6'})
xticks([1 5 9 13 17 21])
xticklabels({'1.0','1.5','2.0','2.5','3.0','3.5'})
xlabel('Significant Wave Height Threshold [m]','Color','k')
ylabel('Required Window Length [h]','Color','k')
lgd = legend({'Observed',m_type});
lgd.Location = 'southeast';
hold off
grid on

% Low Values Normalized Persistence Below
figure(7)
clf
v = [below_values(1),below_values(1)];
[C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
clabel(C,h)
hold on
[C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
clabel(C,h)

v = [below_values(2),below_values(2)];
```

```
    [C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    hold on
    [C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
    clabel(C,h)

    v = [below_values(3),below_values(3)];
    [C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    hold on
    [C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
    clabel(C,h)

    v = [below_values(4),below_values(4)];
    [C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    hold on
    [C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
    clabel(C,h)

    v = [below_values(5),below_values(5)];
    [C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    hold on
    [C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
    clabel(C,h)

    v = [below_values(6),below_values(6)];
    [C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    [C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
    clabel(C,h)

    v = [below_values(7),below_values(7)];
    [C,h] = contour(flipud(hindBelowTime),v,'b','LineWidth',line_width);
    clabel(C,h)
    [C,h] = contour(flipud(simBelowTime),v,'--r','LineWidth',line_width);
    clabel(C,h)
    yticks([1 10 20 30 40 50 60])
    yticklabels({'360','300','240','180','120','60','6'})
    xticks([1 5 9 13 17 21])
    xticklabels({'1.0','1.5','2.0','2.5','3.0','3.5'})
    xlabel('Significant Wave Height Threshold [m]','Color','k')
    ylabel('Required Window Length [h]','Color','k')
    lgd = legend({'Observed',m_type});
    lgd.Location = 'southeast';
    hold off
    grid on
end
```

persistence.m

```
% CAUTION: The function intersections.m behaves strange for some
% threshold values where it results in erroerous results.

function [aboveBin, belowBin] = persistence(series,step_length)
% Length of series
nSeries = length(series);

% Duration of smallest weather windows
binSize = 6;
% Number of weather windows
numBins = 60;
% Number of steps for Hs
%Hs_step = 20;

% Initialize vectors for intersection determination
x1 = 1:nSeries;
x2 = 1:nSeries;
y2(1,:) = series;

% Value vector
valVec = 1:0.25:4;

% Preallocation
belowBin = zeros(numBins,length(valVec));
aboveBin = zeros(numBins,length(valVec));

% Access and Waiting
for kk = 1:length(valVec)

% Treshold vector, 0.999 to avoid counting intersection points
y1 = ones(1,nSeries)*valVec(kk)*0.999;

% Find intersections
[x0,y0] = intersections(x1,y1,x2,y2);
%% ---------------------- Threshold Windows ----------------------------

% ------------------------- Below Treshold -----------------------------
if valVec(kk) <= series(1) % Window starts later in series
    belowTresh = zeros(1,floor(length(x0)/2));
    for ii = 1:1:floor(length(x0)/2)
        belowTresh(ii) = x0(2*ii)-x0(2*ii-1); % Length of WWs
    end
    if valVec(kk) >= series(end)
        belowTresh(end+1) = length(series)-x0(end);
    end
else % Window starts at begining of series
    belowTresh = zeros(1,ceil(length(x0)/2));
    belowTresh(1) = x0(1);
    for ii = 1:ceil(length(x0)/2)-1
        belowTresh(ii+1) = x0(2*ii+1)-x0(2*ii); % Length of WWs
    end
    if valVec(kk) >= series(end)
        belowTresh(end+1) = length(series)-x0(end);
```

```
    end
end

% ------------------------ Above Treshold --------------------------------
if valVec(kk) <= series(1) % Window starts at begining of series
    aboveTresh = zeros(1,ceil(length(x0)/2));
    aboveTresh(1) = x0(1);
    for ii = 1:ceil(length(x0)/2)-1
        aboveTresh(ii+1) = x0(2*ii+1)-x0(2*ii); % Length of WWs
    end
    if valVec(kk) <= series(end)
        aboveTresh(end+1) = length(series)-x0(end);
    end
else % Window starts later in series
    aboveTresh = zeros(1,floor(length(x0)/2));
    for ii = 1:1:floor(length(x0)/2)
        aboveTresh(ii) = x0(2*ii)-x0(2*ii-1); % Length of WWs
    end
    if valVec(kk) <= series(end)
        aboveTresh(end+1) = length(series)-x0(end);
    end
end

%% ----------------- Time Above and Below Threshold -----------------------
% Transform time steps to real time
belowTresh = belowTresh*step_length;
aboveTresh = aboveTresh*step_length;

% Time BELOW Threshold Matrice
for ii = 1:length(belowTresh)
    for jj = 1:numBins
       belowBin(jj,kk)=belowBin(jj,kk)+floor(belowTresh(ii)/(binSize*jj));
    end
end

% Time ABOVE Threshold Matrice
for ii = 1:length(aboveTresh)
    for jj = 1:numBins
       aboveBin(jj,kk)=aboveBin(jj,kk)+floor(aboveTresh(ii)/(binSize*jj));
    end
end
end
end
```

over_all_mean_variance.m

```
function overallResults = over_all_mean_variance(HsObs,TpObs,Ysim,num_rep)

% Over-all mean and variance of observed values
meanObsHs = mean(HsObs);
varObsHs = var(HsObs);
```

```
meanObsTp = mean(TpObs);
varObsTp = var(TpObs);
cov_obs = cov(HsObs,TpObs);
covObs = cov_obs(1,2);

meanSimHs = mean(mean(Ysim(:,1,:)));
varSimHs = mean(var(Ysim(:,1,:)));
meanSimTp = mean(mean(Ysim(:,2,:)));
varSimTp = mean(var(Ysim(:,2,:)));

cov_sum = zeros(2,2);
for ii = 1:num_rep
    cov_syn = cov(Ysim(:,1,ii),Ysim(:,2,ii));
    cov_sum = cov_sum + cov_syn;
end

cov_sim = cov_sum/num_rep;
covSim = cov_sim(1,2);

% Store overall results
overallResults = [meanSimHs varSimHs meanSimTp varSimTp covSim; meanObsHs ...
    varObsHs meanObsTp varObsTp covObs];
end
```

monthly_mean_covariance.m

```
function [obs_cov,sim_cov,obs_month_mean,sim_month_mean]=monthly_mean_covariance(...metocea

% Observed values
HsObs = metocean(:,3);
TpObs = metocean(:,4);
% Initilize
obsMonthMeanHs = zeros(1,12); obsMonthMeanTp = zeros(1,12);
obsCov11 = zeros(1,12);obsCov12 = zeros(1,12);obsCov22 = zeros(1,12);
obsMonthCov = zeros(2,2,12);

% Build matrix/vectors holding mean and cov for each month for observed
for ii = 1:12
    obsMonthMeanHs(ii) = mean(HsObs(metocean(:,2) == ii));
    obsMonthMeanTp(ii) = mean(TpObs(metocean(:,2) == ii));
    obsMonthCov(:,:,ii) = cov(HsObs(metocean(:,2) == ii),...
                    TpObs(metocean(:,2) == ii));
end

% Transform cov matrix into vectors for each element
for ii = 1:12
    obsCov11(ii) = obsMonthCov(1,1,ii);
    obsCov12(ii) = obsMonthCov(1,2,ii);
    obsCov22(ii) = obsMonthCov(2,2,ii);
end
```

```
% Monthly mean and covariances
obs_month_mean = [obsMonthMeanHs;obsMonthMeanTp];
obs_cov = [obsCov11;obsCov12;obsCov22];

% Simulated values
% Initialize
simMonthMeanHsRep = zeros(numRep,12);simMonthMeanTpRep = zeros(numRep,12);
simMonthCovRep = zeros(2,2,12);
simCov11 = zeros(1,12);simCov12 = zeros(1,12);simCov22 = zeros(1,12);

% Build matrix/vectors holding mean and cov for each month for replications
for ii = 1:numRep
    for jj = 1:12
        simMonthMeanHsRep(ii,jj) = mean(Ysim(metocean(:,2) == jj,1,ii));
        simMonthMeanTpRep(ii,jj) = mean(Ysim(metocean(:,2) == jj,2,ii));
        % Adding cov matrices, then dividing due to matrix structure
        simMonthCovRep(:,:,jj) = simMonthCovRep(:,:,jj)+...
            cov(Ysim(metocean(:,2)==jj,1,ii),Ysim(metocean(:,2)==jj,2,ii));
    end
end

% Monthly mean and cov over all replications
sim_month_mean = [mean(simMonthMeanHsRep);mean(simMonthMeanTpRep)];
simMonthCov = simMonthCovRep/numRep;

% Transform cov matrix into vectors for each element
for ii = 1:12
    simCov11(ii) = simMonthCov(1,1,ii);
    simCov12(ii) = simMonthCov(1,2,ii);
    simCov22(ii) = simMonthCov(2,2,ii);
end

sim_cov = [simCov11;simCov12;simCov22];
end
```

intersections.m

```
function [x0,y0,iout,jout] = intersections(x1,y1,x2,y2,robust)
% By  Douglas Schwarz, MATLAB File Exchange

%INTERSECTIONS Intersections of curves.
%   Computes the (x,y) locations where two curves intersect.  The curves
%   can be broken with NaNs or have vertical segments.
%
% Example:
%   [X0,Y0] = intersections(X1,Y1,X2,Y2,ROBUST);
%
% where X1 and Y1 are equal-length vectors of at least two points and
% represent curve 1.  Similarly, X2 and Y2 represent curve 2.
% X0 and Y0 are column vectors containing the points at which the two
% curves intersect.
```

```
%
% ROBUST (optional) set to 1 or true means to use a slight variation of the
% algorithm that might return duplicates of some intersection points, and
% then remove those duplicates.  The default is true, but since the
% algorithm is slightly slower you can set it to false if you know that
% your curves don't intersect at any segment boundaries.  Also, the robust
% version properly handles parallel and overlapping segments.
%
% The algorithm can return two additional vectors that indicate which
% segment pairs contain intersections and where they are:
%
%   [X0,Y0,I,J] = intersections(X1,Y1,X2,Y2,ROBUST);
%
% For each element of the vector I, I(k) = (segment number of (X1,Y1)) +
% (how far along this segment the intersection is).  For example, if I(k) =
% 45.25 then the intersection lies a quarter of the way between the line
% segment connecting (X1(45),Y1(45)) and (X1(46),Y1(46)).  Similarly for
% the vector J and the segments in (X2,Y2).
%
% You can also get intersections of a curve with itself.  Simply pass in
% only one curve, i.e.,
%
%   [X0,Y0] = intersections(X1,Y1,ROBUST);
%
% where, as before, ROBUST is optional.

% Version: 2.0, 25 May 2017
% Author:  Douglas M. Schwarz
% Email:   dmschwarz=ieee*org, dmschwarz=urgrad*rochester*edu
% Real_email = regexprep(Email,{'=','*'},{'@','.'})

% Input checks.
if verLessThan('matlab','7.13')
    error(nargchk(2,5,nargin)) %#ok<NCHKN>
else
    narginchk(2,5)
end

% Adjustments based on number of arguments.
switch nargin
    case 2
        robust = true;
        x2 = x1;
        y2 = y1;
        self_intersect = true;
    case 3
        robust = x2;
        x2 = x1;
        y2 = y1;
        self_intersect = true;
    case 4
        robust = true;
        self_intersect = false;
    case 5
```

```
        self_intersect = false;
end

% x1 and y1 must be vectors with same number of points (at least 2).
if sum(size(x1) > 1) ~= 1 || sum(size(y1) > 1) ~= 1 || ...
        length(x1) ~= length(y1)
    error('X1 and Y1 must be equal-length vectors of at least 2 points.')
end
% x2 and y2 must be vectors with same number of points (at least 2).
if sum(size(x2) > 1) ~= 1 || sum(size(y2) > 1) ~= 1 || ...
        length(x2) ~= length(y2)
    error('X2 and Y2 must be equal-length vectors of at least 2 points.')
end

% Force all inputs to be column vectors.
x1 = x1(:);
y1 = y1(:);
x2 = x2(:);
y2 = y2(:);

% Compute number of line segments in each curve and some differences we'll
% need later.
n1 = length(x1) - 1;
n2 = length(x2) - 1;
xy1 = [x1 y1];
xy2 = [x2 y2];
dxy1 = diff(xy1);
dxy2 = diff(xy2);

% Select an algorithm based on MATLAB version and number of line
% segments in each curve.
if n1 > 1000 || n2 > 1000 || verLessThan('matlab','7.4')
    % Determine which curve has the most line segments.
    if n1 >= n2
        % Curve 1 has more segments, loop over segments of curve 2.
        ijc = cell(1,n2);
        min_x1 = mvmin(x1);
        max_x1 = mvmax(x1);
        min_y1 = mvmin(y1);
        max_y1 = mvmax(y1);
        for k = 1:n2
            k1 = k + 1;
            ijc{k} = find( ...
                min_x1 <= max(x2(k),x2(k1)) & max_x1 >= min(x2(k),x2(k1)) & ...
                min_y1 <= max(y2(k),y2(k1)) & max_y1 >= min(y2(k),y2(k1)));
            ijc{k}(:,2) = k;
        end
        ij = vertcat(ijc{:});
        i = ij(:,1);
        j = ij(:,2);
    else
        % Curve 2 has more segments, loop over segments of curve 1.
        ijc = cell(1,n1);
        min_x2 = mvmin(x2);
```

```
        max_x2 = mvmax(x2);
        min_y2 = mvmin(y2);
        max_y2 = mvmax(y2);
        for k = 1:n1
            k1 = k + 1;
            ijc{k}(:,2) = find( ...
                min_x2 <= max(x1(k),x1(k1)) & max_x2 >= min(x1(k),x1(k1)) & ...
                min_y2 <= max(y1(k),y1(k1)) & max_y2 >= min(y1(k),y1(k1)));
            ijc{k}(:,1) = k;
        end
        ij = vertcat(ijc{:});
        i = ij(:,1);
        j = ij(:,2);
    end

elseif verLessThan('matlab','9.1')
    % Use bsxfun.
    [i,j] = find( ...
        bsxfun(@le,mvmin(x1),mvmax(x2).') & ...
        bsxfun(@ge,mvmax(x1),mvmin(x2).') & ...
        bsxfun(@le,mvmin(y1),mvmax(y2).') & ...
        bsxfun(@ge,mvmax(y1),mvmin(y2).'));

else
    % Use implicit expansion.
    [i,j] = find( ...
        mvmin(x1) <= mvmax(x2).' & mvmax(x1) >= mvmin(x2).' & ...
        mvmin(y1) <= mvmax(y2).' & mvmax(y1) >= mvmin(y2).');

end


% Find segments pairs which have at least one vertex = NaN and remove them.
if self_intersect
    remove = isnan(sum(dxy1(i,:) + dxy2(j,:),2)) | j <= i + 1;
else
    remove = isnan(sum(dxy1(i,:) + dxy2(j,:),2));
end
i(remove) = [];
j(remove) = [];

% Initialize matrices.  We'll put the T's and B's in matrices and use them
% one column at a time.  AA is a 3-D extension of A where we'll use one
% plane at a time.
n = length(i);
T = zeros(4,n);
AA = zeros(4,4,n);
AA([1 2],3,:) = -1;
AA([3 4],4,:) = -1;
AA([1 3],1,:) = dxy1(i,:).';
AA([2 4],2,:) = dxy2(j,:).';
B = -[x1(i) x2(j) y1(i) y2(j)].';

% Loop through possibilities.  Trap singularity warning and then use
```

```
% lastwarn to see if that plane of AA is near singular.  Process any such
% segment pairs to determine if they are colinear (overlap) or merely
% parallel.  That test consists of checking to see if one of the endpoints
% of the curve 2 segment lies on the curve 1 segment.  This is done by
% checking the cross product
%
%   (x1(2),y1(2)) - (x1(1),y1(1)) x (x2(2),y2(2)) - (x1(1),y1(1)).
%
% If this is close to zero then the segments overlap.

% If the robust option is false then we assume no two segment pairs are
% parallel and just go ahead and do the computation.  If A is ever singular
% a warning will appear.  This is faster and obviously you should use it
% only when you know you will never have overlapping or parallel segment
% pairs.

if robust
    overlap = false(n,1);
    warning_state = warning('off','MATLAB:singularMatrix');
    % Use try-catch to guarantee original warning state is restored.
    try
        lastwarn('')
        for k = 1:n
            T(:,k) = AA(:,:,k)\B(:,k);
            [unused,last_warn] = lastwarn; %#ok<ASGLU>
            lastwarn('')
            if strcmp(last_warn,'MATLAB:singularMatrix')
                % Force in_range(k) to be false.
                T(1,k) = NaN;
                % Determine if these segments overlap or are just parallel.
                overlap(k) = rcond([dxy1(i(k),:);xy2(j(k),:) - xy1(i(k),:)]) < eps;
            end
        end
        warning(warning_state)
    catch err
        warning(warning_state)
        rethrow(err)
    end
    % Find where t1 and t2 are between 0 and 1 and return the corresponding
    % x0 and y0 values.
    in_range = (T(1,:) >= 0 & T(2,:) >= 0 & T(1,:) <= 1 & T(2,:) <= 1).';
    % For overlapping segment pairs the algorithm will return an
    % intersection point that is at the center of the overlapping region.
    if any(overlap)
        ia = i(overlap);
        ja = j(overlap);
        % set x0 and y0 to middle of overlapping region.
        T(3,overlap) = (max(min(x1(ia),x1(ia+1)),min(x2(ja),x2(ja+1))) + ...
            min(max(x1(ia),x1(ia+1)),max(x2(ja),x2(ja+1)))).'/2;
        T(4,overlap) = (max(min(y1(ia),y1(ia+1)),min(y2(ja),y2(ja+1))) + ...
            min(max(y1(ia),y1(ia+1)),max(y2(ja),y2(ja+1)))).'/2;
        selected = in_range | overlap;
    else
        selected = in_range;
```

```
    end
    xy0 = T(3:4,selected).';

    % Remove duplicate intersection points.
    [xy0,index] = unique(xy0,'rows');
    x0 = xy0(:,1);
    y0 = xy0(:,2);

    % Compute how far along each line segment the intersections are.
    if nargout > 2
        sel_index = find(selected);
        sel = sel_index(index);
        iout = i(sel) + T(1,sel).';
        jout = j(sel) + T(2,sel).';
    end
else % non-robust option
    for k = 1:n
        [L,U] = lu(AA(:,:,k));
        T(:,k) = U\(L\B(:,k));
    end

    % Find where t1 and t2 are between 0 and 1 and return the corresponding
    % x0 and y0 values.
    in_range = (T(1,:) >= 0 & T(2,:) >= 0 & T(1,:) < 1 & T(2,:) < 1).';
    x0 = T(3,in_range).';
    y0 = T(4,in_range).';

    % Compute how far along each line segment the intersections are.
    if nargout > 2
        iout = i(in_range) + T(1,in_range).';
        jout = j(in_range) + T(2,in_range).';
    end
end

% Plot the results (useful for debugging).
% plot(x1,y1,x2,y2,x0,y0,'ok');

function y = mvmin(x)
% Faster implementation of movmin(x,k) when k = 1.
y = min(x(1:end-1),x(2:end));

function y = mvmax(x)
% Faster implementation of movmax(x,k) when k = 1.
y = max(x(1:end-1),x(2:end));
```