Automatic Control
Mechatronics

Max Schüssler

Machine Learning with Nonlinear State Space Models

Dissertation

Schriftenreihe der Arbeitsgruppe
Mess- und Regelungstechnik – Mechatronik
Department Maschinenbau

Herausgeber Oliver Nelles

UNIVERSITÄT
SIEGEN

Band 7

Impressum
Prof. Dr.-Ing. Oliver Nelles
Arbeitsgruppe Mess- und Regelungstechnik
Department Maschinenbau
Universität Siegen
57076 Siegen
ISSN 2193-0538
URN urn:nbn:de:hbz:467-22533
Zugl.: Siegen, Univ., Diss., 2022

Machine Learning with Nonlinear State Space Models

DISSERTATION
zur Erlangung des Grades eines
Doktors der Ingenieurwissenschaften

vorgelegt von
Max Schüssler, M.Sc.
aus Bad Berleburg

genehmigt durch die Naturwissenschaftlich-Technische Fakultät
der Universität Siegen
Siegen 2022

Betreuer und erster Gutachter
Prof. Dr.-Ing. Oliver Nelles
Universität Siegen

Zweiter Gutachter
Prof. Dr.-Ing. Peter Kraemer
Universität Siegen

Tag der mündlichen Prüfung
28.07.2022

Acknowledgments

This dissertation was written during my time as a research associate at the Institute of Mechanics and Control Engineering - Mechatronics at the University of Siegen. First and foremost, I would like to express my gratitude and appreciation to my doctoral advisor Prof. Dr.-Ing. Oliver Nelles for his guidance and support. His eagerness for precise yet intuitive explanations and the working environment created by his leadership significantly contributed to the success of this work.

I would also like to thank Prof. Dr.-Ing. Peter Kraemer for his sincere interest in my work and the supervision of this dissertation as a second examiner.

The valuable cooperations with Technip Zimmer GmbH, Viega GmbH & Co. KG, Deutz AG, and others allowed me to apply system identification and machine learning methods to real-world problems. I deeply appreciated those opportunities and appealing projects.

A pleasant working atmosphere is nothing that can be taken for granted. I thank all my colleagues that made this journey so exciting with their inspiring ideas, support, and many on- and off-topic discussions. Special thanks belongs to Diana Klein, Geritt Kampmann, Tobias Münker, Tim Oliver Heinz, Timm Julian Peter, Tim Decker, Christopher Illg, Tarek Kösters, and Fabian Schneider. I would also like to express my thanks to Benedikt Wüllner, who eagerly helped and supported this research project as a student assistant.

I want to thank my parents, Christian and Claudia, for their unconditional support. They helped me develop a curious mindset and fostered the desire to get to the bottom of things. Last but not least, this whole endeavor would not have been possible without my much-loved wife Lea and our kids, Johann and Edith. Lea's enduring love and patience helped me tremendously to set priorities and focus on the things that are most vital in life.

Hilchenbach, August 2022
Max Schüssler

Abstract

Contemporary automation systems require accurate models for analysis, design, and control. Oftentimes, it is not possible to derive satisfactory models by first principles. This is often the case in process engineering or in mechatronic systems, where the overall process is just too complex, or the development of those first-principle models would just be too expensive. Thus, there is a strong demand for data-driven modeling approaches. Additionally, there are enormous successes in machine learning concerning the generation of models solely by the use of data.

Driven by the need for accurate models as well as the successes and advances in machine learning, a novel class of model structures and associated training algorithms for building data-driven nonlinear dynamic models is developed. The new identification procedure and the resulting model is called local model state space network (LMSSN). It fuses nonlinear state space models with local model networks (LMNs). The LMSSN is designed as a user-friendly and deterministic approach with warm-start initialization (with a linear model), incrementally growing complexity, favorable and robust extrapolation behavior, easy interpretation, and straightforward incorporation possibilities for prior knowledge.

Furthermore, recurrent neural networks (RNNs) and their similarities to nonlinear state space models are elaborated on. Popular RNNs like the long short-term memory (LSTM) and gated recurrent unit (GRU) models are translated into controls perspective and a comprehensive study comparing different RNN structures is carried out.

The overall outstanding performance of the LMSSN is demonstrated on various applications. It is shown that the LMSSN can accurately model a wide variety of processes by consistently computing expressive yet compact models. The LMSSN is benchmarked against state-of-the-art nonlinear system identification algorithms and achieves similar or superior results. The practical usefulness and applicability of LMSSN are strikingly demonstrated on two real-world processes.

Kurzfassung

Genaue Modelle sind die Voraussetzung für die Analyse, Auslegung und Regelung von modernen Automatisierungssystemen. Allerdings können genaue Modelle oft nicht durch physikalische Modellbildung entwickelt werden. Dies gestaltet sich insbesondere in der Verfahrenstechnik und für mechatronische Systeme als sehr herausfordernd, da die Gesamtprozesse oft zu komplex oder die Entwicklung eines physikalischen Modells oft zu teuer ist. Datengetriebene Modelle gewinnen deshalb zunehmend an Bedeutung. Dieser Trend wird außerdem durch Erfolge im Bereich des maschinellen Lernens stark begünstigt.

Aufgrund des starken Bedarfs an genauen Modellen und motiviert durch die Erfolge des maschinellen Lernens wird eine neue Klasse von Modellstrukturen und dazugehörigen Trainingsalgorithmen für die datengetriebene Modellierung von nichtlinearen dynamischen Prozessen entwickelt. Der neue Identifikationsalgorithmus und das resultierende Modell heißen *Local Model State Space Network* (LMSSN). Es vereint nichtlineare Zustandsraummodelle mit lokalen Modellnetzen (LMNs). LMSSN ist ein benutzerfreundlicher und deterministischer Algorithmus, der mit einem linearen Modell initialisiert wird (Warmstart). LMSSN hat inkrementell steigende Komplexität, ein gewünschtes und robustes Extrapolationsverhalten, eine einfache Interpretierbarkeit und bietet unkomplizierte Möglichkeiten zur Einbringung von Vorwissen.

Außerdem werden rekurrente neuronale Netze (RNNs) und deren Ähnlichkeit zu nichtlinearen Zustandsraummodellen behandelt. Beliebte RNNs wie das *Long Short-Term Memory* (LSTM) und *Gated Recurrent Unit* (GRU) Modell werden aus einer regelungstechnischen Perspektive veranschaulicht und unterschiedliche RNN-Strukturen werden umfassend in einer Simulationsstudie verglichen.

Die insgesamt herausragende Leistungsfähigkeit von LMSSN wird anhand einiger Anwendungsbeispiele demonstriert. Eine Vielzahl unterschiedlicher Prozesse können mit LMSSN sehr gut modelliert werden, wobei stets aussagekräftige, aber dennoch kompakte Modelle generiert werden. Die LMSSN-Methode wird mit anderen modernen Algorithmen zur nichtlinearen Systemidentifikation verglichen. Hier liefert LMSSN gleichwertige, wenn nicht sogar bessere Ergebnisse. Die praktische Anwendbarkeit wird eindrucksvoll an zwei industriellen Prozessen veranschaulicht.

Contents

Symbols and Acronyms

Notation

$\underline{X}, \underline{Y}, \underline{Z}$	matrices
$\underline{x}, \underline{y}, \underline{z}$	vectors
x, y, z	scalars
$X_{i,j}$	element in i-th row and j-th colum of matrix $\underline{X}$
x_i	i-th element of vector $\underline{x}$
$\frac{\partial f(\underline{x})}{\partial x_i}$	partial derivative of $f(\underline{x})$ with respect to x_i
$\dot{x}$	first derivative with respect to time
$\ddot{x}$	second derivative with respect to time
$\hat{(\cdot)}$	estimation, modeled quantity
$\bar{(\cdot)}$	mean
$(\cdot)^T$	transpose of matrix or vector
$(\cdot)^{-1}$	inverse function or matrix
$(\cdot)^{[s]}$	quantity belonging to state equation of state space model
$(\cdot)^{[o]}$	quantity belonging to output equation of state space model
$\lVert\cdot\rVert_1$	L_1-norm
$\odot$	element-wise product (Hadamard product)
$\oslash$	element-wise division (Hadamard division)
$\otimes$	Kronecker product

Latin Symbols

$\underline{A}$	state matrix of state space model
$\underline{b}$	input vector of state space model
$\underline{c}^T$	output vector of state space model
d	feedthrough of state space model

$\mathbb{E}$	expectation operator
$\underline{E}$	polynomial coefficient matrix of state equation of PNLSS
$e(k)$	error at time step k
f_s	sampling frequency
$\underline{f}^T$	polynomial coefficient vector of output equation of PNLSS
$f(\cdot)$	(nonlinear) function
$G(q)$	linear transfer function
$\underline{g}$	gradient
$g(\cdot)$	output equation of state space model
$\underline{H}$	Hessian matrix
$\underline{h}(\cdot)$	state equation
I	loss function
$\underline{I}$	identity matrix
$\underline{J}$	Jacobian matrix
k	discrete time step
k_σ	proportionality factor for width of radial basis functions
L	local model
m	order of external dynamics model
N	number of data samples
n_m	number of local models in state equation of state space model
n_o	number of local models in output equation of state space model
n_p	number of inputs
n_q	number of outputs
n_x	number of state variables, order of state space model
n_θ	number of parameters
$n(k)$	noise at time step k
$\mathcal{O}(\underline{\zeta})$	degrees of monomials in PNLSS state equation
$\mathcal{O}(\underline{\eta})$	degrees of monomials in PNLSS output equation
$\underline{o}$	offset in state equation of state space model
p	offset in output equation of state space model
q	forward-shift operator, i.e., $q^{-1}x(k) = x(k-1)$
$\mathbb{R}$	real numbers
$\underline{R}$	rotation and scaling matrix for gradient
$\underline{s}$	input space to local models
$\underline{T}$	transformation matrix
T_s	sampling time
$\underline{t}$	offset vector of affine transformation

$u(k)$	process or model input at time step k
$\underline{\tilde{u}}(k)$	inner input vector $\underline{\tilde{u}}(k) = [\underline{\hat{x}}^T(k)\ u(k)]^T$
$\underline{v}$	sigmoid split parameters
$\underline{\hat{x}}(k)$	state vector at time step k
$\underline{\hat{x}}_0$	initial state vector
$y(k)$	process output at time step k
$\hat{y}(k)$	model output at time step k
$\underline{z}$	input space of validity functions
z	time-shift operator q in the frequency domain

Greek Symbols

η	step size
$\underline{\eta}(\cdot)$	monomials in output equation of PNLSS
κ	steepness parameter for sigmoid functions
$\underline{\mu}$	center coordinates of radial basis function
$\varPhi$	validity function of local model network
$\varPsi$	membership function of local model network
$\underline{\Sigma}$	squared standard deviations of radial basis function (matrix)
$\underline{\sigma}$	standard deviations of radial basis function (vector)
$\underline{\Theta}$	parameter matrix
$\underline{\theta}$	parameter vector
$\underline{\zeta}(\cdot)$	monomials in state equation of PNLSS

Acronyms

ADAGRAD	adaptive gradient algorithm
ADAM	adaptive moment estimation
AIC$_\mathrm{c}$	corrected Akaike information criterion
APRBS	amplitude-modulated pseudo random binary signal
ARX	autoregressive with exogenous input
BIC$_\mathrm{c}$	corrected Bayesian information criterion
BLA	best linear approximation
BPTT	backpropagation-through-time
DDLC	data-driven local coordinates

DFT	discrete Fourier transform
FIR	finite impulse response
FRF	frequency response function
GP	Gaussian process
GRU	gated recurrent unit
HILOMOT	hierarchical local model tree
LLM	local linear model
LM	local model
LMN	local model network
LMSSN	local model state space network
LOLIMOT	local linear model tree
LPV	linear parameter varying
LSTM	long short-term memory
LTI	linear time-invariant
MIMO	multiple-input multiple-output
MISO	multiple-input single-output
MLP	multilayer perceptron
NARMAX	nonlinear autoregressive moving average with exogenous input
NARX	nonlinear autoregressive with exogenous input
NFIR	nonlinear finite impulse response
NOBF	nonlinear orthonormal basis function
NOE	nonlinear output error
NRBF	normalized radial basis function
NRMSE	normalized root mean squared error
NRTC	non-road transient cycle
ODE	ordinary differential equation
OE	output error
OMNIPUS	optimized nonlinear input signal
PDF	probability density function
PNLSS	polynomial nonlinear state space model
PWA	piecewise affine
RBF	radial basis function
ReLU	rectified linear unit
rms	root mean squared
RMSE	root mean squared error
RMSprop	root mean square propagation
RNN	recurrent neural network

SGD	stochastic gradient descent
SISO	single-input single-output
SNR	signal-to-noise ratio
SVD	singular value decomposition
TCN	temporal convolutional network
WLM	weighted local model

1 Introduction

Data-driven modeling is the research field of gathering knowledge about a system by drawing conclusions from that system's input and output data. A system in this manner can be any kind of object in which variables of different kinds interact [69]. The inputs can be external stimuli or disturbances, while the outputs are the observable outcome of interest.

Another, more widespread, term for data-driven modeling is *machine learning*. Recent successes in natural language processing [12] and computer vision [51, 22] have led to a greatly expanded interest of academia and industry in this field. Since many technical plants and processes can be described in a system-oriented framework, the application of machine learning methods to build models (mathematical description) of those processes is of great interest [70].

Another reason to use data-driven approaches is that there are many fields of engineering in which it is difficult to obtain satisfactory models by first principles. This is often the case in process engineering or in mechatronic systems, where the overall process is just too complex. Thus, there is a strong demand for data-driven modeling approaches.

1.1 Two Different Perspectives

Data-driven models can be distinguished by their behavior but also by their origin.

Model Behavior Perspective The behavior of data-driven approaches can be divided into either being static or dynamic, and on the other hand, into being linear or nonlinear (Fig. 1.1).

Static approaches may be employed whenever the process can be modeled without temporal dependencies. The model of the process is, therefore, governed by algebraic

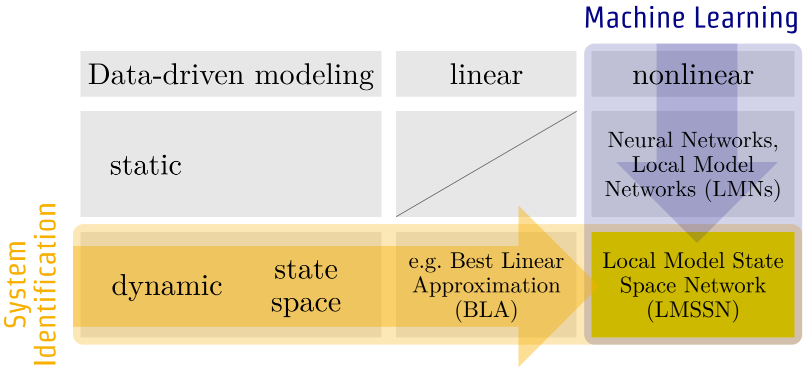

Machine Learning
Data-driven modeling
linear
nonlinear
static
Neural Networks, Local Model Networks (LMNs)
System Identification
dynamic
state space
e.g. Best Linear Approximation (BLA)
Local Model State Space Network (LMSSN)

Figure 1.1: Data-driven modeling

equations. Contrarily, if the process has temporal dependencies with respect to its inputs and outputs, the process is described with a dynamic model. In terms of physical modeling, this means that differential equations are employed.

The class of linear models encompasses only models which depend linearly on their inputs. The class of nonlinear models comprises all models *except* linear models. Therefore, this class is much harder to grasp as an infinite variety of model architectures exits.

The controls community developed over quite some time data-driven linear models for dynamic systems, which is known by the term *system identification*. On the other hand, the computer science and machine learning communities developed models mostly for nonlinear static applications. Combining those two fields yields the field of *nonlinear system identification*. It is pointed out in [68] that even though the general field of system identification is quite a mature one, there are nevertheless some open areas in system identification: among those maybe the most important one being nonlinear system identification. This research field is becoming increasingly important [130, 69, 89] to account for dominant nonlinear behavior in processes and because of the need to fulfill increasing performance requirements.

Origin Perspective The machine learning community does not only deal with static processes but also proposes structures for dynamic processes such as recurrent neural networks (RNNs). It turns out that those structures are, in their essence, an equivalent structure to nonlinear state space models (see Fig. 1.2) which originate from the system identification community.

However, the developments in both fields have been surprisingly isolated, with journals and conferences on their own [68]. This development has just recently been

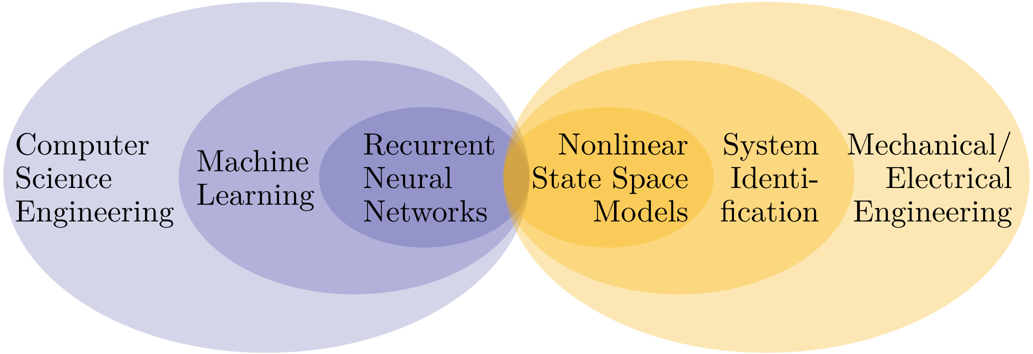


Figure 1.2: Data-driven modeling from different perspectives

broken, with more and more researchers trying to bring the two fields, machine learning and system identification, together [70, 123]. This shows that both fields intersect in nonlinear system identification in two ways. First, the static nonlinear methods from the machine learning world can be combined with the nonlinear state space model from the system identification community. Second, both fields have worked on dynamics representations, but it is not always clear how specific methods translate into the other world.

1.2 Objectives and Own Contribution

The objective of this thesis is the development of a novel nonlinear system identification algorithm that combines elements from the machine learning and system identification field, see Fig. 1.1. It focuses from the dynamical perspective on nonlinear state space models (one possibility of a dynamics realization) and from the static perspective on local model networks (LMNs) (modeling nonlinearities). We call the novel identification procedure local model state space network (LMSSN).

Especially in engineering applications, it is important that methods stay as simple as possible and allow for interpretability. Hence, divide-and-conquer strategies are liked by engineers since a very complex problem can be tackled by dealing with multiple simpler problems. Multiple model approaches with LMNs, which form one fundamental building block of the LMSSN, follow exactly this line of thinking.

Originally, system identification only referred to black-box modeling which means no prior knowledge is used for modeling. In contrast, white-box modeling is solely based on physical insight and first-principle knowledge. One basic rule in estimation is *not to estimate what you already know* [130]. Therefore, system identification nowadays encompasses a whole *palette of gray shades*, depending on how much prior knowledge is incorporated into the model [68, 117]. The LMSSN is a black-box model at its core, which nevertheless allows for the incorporation of prior knowledge if available. Thus, the *shade* of the model can be shifted from black to gray.

There is also a lot of research done in system identification in a stochastic framework [141, 114, 7, 40]. This thesis nevertheless focuses solely on deterministic approaches to make the derived methodologies more accessible. It has also been demonstrated that errors due to an inaccurate description of the nonlinear functions surpasses the error due to unsuitable noise models [118].

This thesis contains the following own original work:

Local Model State Space Network A novel class of model structures and associated training algorithms for building data-driven nonlinear state space models is developed. The LMSSN is designed as user-friendly and deterministic approach with warm-start initialization (with a linear model), incrementally growing complexity, favorable and robust extrapolation behavior, easy interpretation, and straightforward incorporation possibilities for prior knowledge. The following aspects shall be emphasized:

- Transformation of the state trajectory to make splits of LMNs within state dimensions possible.
- Strategies for the incorporation of prior knowledge into the LMSSN structure. Similarities of the LMSSN to piecewise affine (PWA) state space models and block-oriented structures are analyzed.
- Numerical issues with normalized radial basis functions (NRBFs) in recurrent structures.
- User-friendly object-oriented LMSSN toolbox with few necessary hyperparameter choices.
- Demonstration of usefulness of the LMSSN on artificial test systems, benchmark problems, and real-world processes.
- Derivation of analytical gradients of the LMSSN with respect to the model parameters.

Recurrent Neural Networks from Controls Perspective The similarities between nonlinear state space models and RNNs are analyzed. Common RNN structures like the long short-term memory (LSTM) and gated recurrent unit (GRU) model are transferred into controls perspective. A comprehensive study regarding various RNN structures is carried out.

Fundamentals Additionally, within Chap. 2 the following sections go beyond the state of the art:

- Analysis and comparison of advantages and disadvantages of internal versus external dynamics approaches.
- Influence of gradient updating frequency on optimization.

1.3 Structure of this Thesis

The structure of this thesis is as follows.

Chapter 2 provides the reader with the theoretical foundations of nonlinear system identification. Different internal and external dynamics realizations, neural network architectures and LMNs, nonlinear optimization strategies, as well as different aspects regarding model complexity, are covered here.

Chapter 3 describes the LMSSN. It is a novel class of model structures and associated training algorithms for building data-driven nonlinear state space models. First, the model structure and construction algorithm are explained. Then, some pivotal steps in LMSSN identification are covered, followed by the characteristics of the model. Finally, relationships to other model structures are explained and some numerical and computational aspects are highlighted.

Chapter 4 presents deep recurrent neural networks. First, the relationship between state space models and RNNs is explained, followed by a detailed description of the building blocks of deep RNNs, including the LSTM and GRU model explained from a controls perspective. The chapter closes with a case study of different deep RNN structures.

Chapter 5 contains the application of LMSSN to artificial test processes, selected benchmark problems, and real-world processes. The LMSSN and its performance are studied and compared to other nonlinear system identification algorithms.

Chapter 6 draws conclusions and gives an outlook on topics for further research.

2 Nonlinear System Identification

The term *system identification* was coined in the 1950s by [147]. It originates from the control community and is used to describe the problem of identifying a black-box model solely by measurements of input and output data of a dynamical system. Those models can be employed for tasks such as simulation, optimization, fault detection/diagnosis, model-based control, and more. Figure 2.1 illustrates the utilization of models for four exemplary tasks.

The general problem of system identification is considered in Fig. 2.2. The inputs $\underline{u}(k) = [u_1(k)\, u_2(k) \ldots u_{n_p}(k)]^T$ enter a process, which is possibly disturbed by noise $n(k)$. The output of the process is denoted by $y(k)$. The same input $\underline{u}(k)$ that excites the process also enters a model, producing the model output $\hat{y}(k)$. The difference between process and model output, the error $e(k) = y(k) - \hat{y}(k)$, is then used to adapt the model so that it resembles the process in the "best possible way".

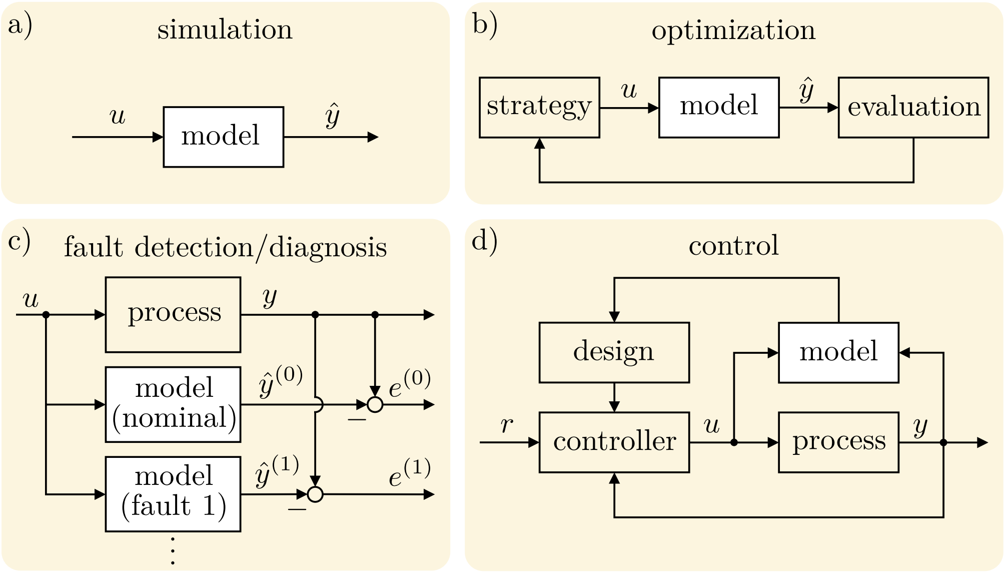


Figure 2.1: Utilization of models for a) simulation, b) optimization, c) fault detection/diagnosis, and d) control [89]. Here, u is the input signal, $\hat{y}$ the model output signal, y the process output signal, e the error signal, and r is the reference signal.

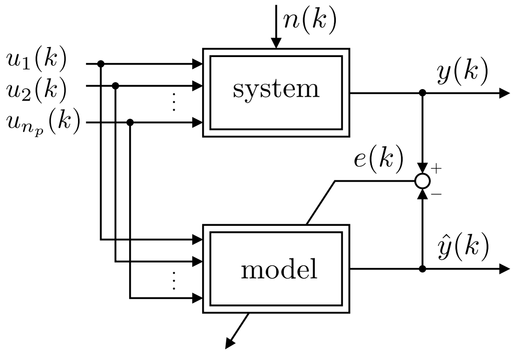


Figure 2.2: Identification of an unknown system (see also [89])

Note that the variable k denotes the discrete time step. The link between the continuous time t and discrete time k is $t = kT_s$ with the sampling time T_s. The data that is used for identification has usually been collected beforehand and is stored in input-output tuples as $\{\underline{u}(k),\ y(k)\}_{k=1}^{N}$, where N is the total number of data points or data samples. Note that the terms process and system are used interchangeably throughout this thesis.

Only discrete-time models are considered here. This choice is motivated by the fact that measurement data is collected by sampling, making relations between discrete-time data and discrete-time models more straightforward [69]. Additionally, when looking at control applications, discrete-time descriptions are more suitable since control actions are usually taken at discrete points in time. Moreover, the estimation of nonlinear continuous-time models is not a trivial task and can be computationally highly complex [98].

Also, note that the depicted system and model have a multiple-input single-output (MISO) form and not the most general multiple-input multiple-output (MIMO) form. This is since MIMO systems and models are usually decomposed to n_q different MISO models. After all, they are easier to understand, handle, and validate than MIMO models [89]. Thus, for simplicity, only MISO and single-input single-output (SISO) systems and models are considered for this thesis.

When talking about *nonlinear* system identification, one challenge is the tremendous scope of this class of models. It is like talking about non-elephant zoology – all animals are considered except for elephants. Likewise, all models are nonlinear models except for the *linear* ones. Therefore, we will focus our scope on models and techniques, which are essential for understanding the here chosen approach, rather than giving a comprehensive account of all existing nonlinear dynamic models that have been developed throughout the years.

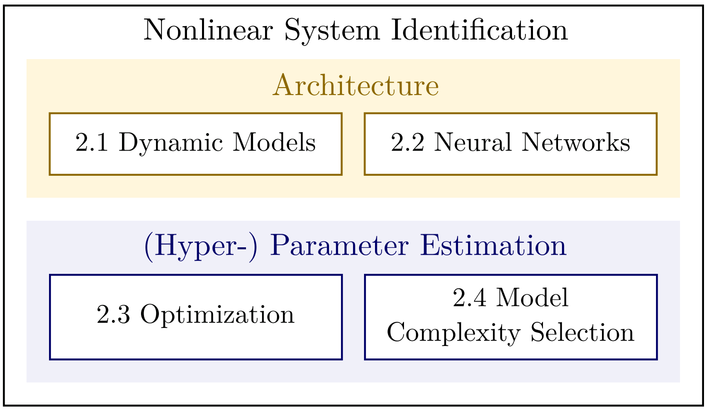

Nonlinear System Identification
Architecture
2.1 Dynamic Models
2.2 Neural Networks
(Hyper-) Parameter Estimation
2.3 Optimization
2.4 Model Complexity Selection

Figure 2.3: Most important system identification topics to be covered in this chapter.

The covered topics of nonlinear system identification are shown in Fig. 2.3. Regarding the architecture of the models, we will start with the topic of dynamic models in Sect. 2.1. Here, different linear and nonlinear as well as external and internal dynamics models will be covered. Then, Sect. 2.2 describes different neural network structures, which can then be employed for nonlinear dynamic modeling. Once a model architecture is chosen, the parameters of the models need to be optimized and certain model complexity choices (hyperparameters) need to be made. Those topics regarding parameter estimation will be dealt with in Sect. 2.3 and Sect. 2.4.

2.1 Dynamic Models

Dynamic models are needed for processes that are describable by differential equations. In the discrete-time context, differential equations can be translated into difference equations, in which the dynamics become visible through the dependence of the current output on past input and output values. If this dependency is modeled explicitly, the arising models are called external dynamics models (or input-output models). On the other hand, if an inner memory or state of the model is introduced, those models are called internal dynamics models (or state space models).

2.1.1 External Dynamics Models

We will first cover external dynamics models. Here, we will start with linear external dynamics models and then cover some nonlinear external dynamics models.

2.1.1.1 Linear External Dynamics Models

In general, the linear modeling framework is well established [64, 69, 89, 101] and therefore linear models are a prevalent choice in a vast range of applications [84, 99]. A general realization for deterministic linear external dynamics SISO models is described by

$$\hat{y}(k) = G(q)u(k) = \frac{B(q)}{\tilde{A}(q)}u(k)\,. \tag{2.1}$$

The model output $\hat{y}(k)$ at the discrete time step k is computed by filtering the input $u(k)$ with a linear filter $G(q)$. Variable q stands for the forward-shift operator in the time domain, i.e., $q^{-1}x(k) = x(k-1)$.

In addition to the deterministic component, a stochastic part might influence the system. This stochastic term can be modeled by filtering white Gaussian noise $v(k)$ with a linear filter $H(q)$ to obtain

$$n(k) = H(q)v(k) = \frac{C(q)}{\tilde{D}(q)}v(k)\,. \tag{2.2}$$

Combining deterministic and stochastic parts delivers a general linear model as

$$\hat{y}(k) = \frac{B(q)}{\tilde{A}(q)}u(k) + \frac{C(q)}{\tilde{D}(q)}v(k)\,. \tag{2.3}$$

It is helpful to separate any possible common denominator dynamics $A(q)$ from $G(q)$ and $H(q)$ for further analysis. Thus, $F(q)A(q) = \tilde{A}(q)$ and $D(q)A(q) = \tilde{D}(q)$ lead to

$$\hat{y}(k) = \frac{B(q)}{F(q)A(q)}u(k) + \frac{C(q)}{D(q)A(q)}v(k)\,. \tag{2.4}$$

If the denominators do not share a common dynamic, $A(q)$ turns to $A(q) = 1$. This general linear dynamic model is shown in Fig. 2.4.

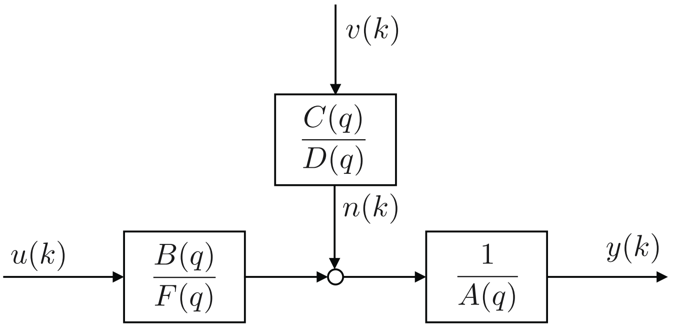


Figure 2.4: General linear dynamic model (see also [89])

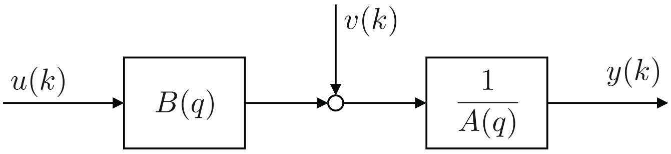


Figure 2.5: ARX model structure (see also [89])

This general model is not used in practice with all possible numerator and denominator terms – widely applied dynamics realizations are obtained by simplification. Two very important models will be explained in more detail.

ARX The probably most commonly used linear model is the autoregressive with exogenous input (ARX) model (Fig. 2.5). It can be derived from the general model equation (2.4) by setting the polynomials $F(q)$, $C(q)$ and $D(q)$ to 1, which leads to

$$A(q)y(k) = B(q)u(k) + v(k)\,. \tag{2.5}$$

The term *autoregressive* refers to the shared denominator dynamics $1/A(q)$ for the deterministic and stochastic part. With this term, previous output values are fed back and an autoregressive structure arises. The X in ARX indicates the *eXogenous input* $u(k)$. This terminology originally comes from the time-series analysis, where only the white noise $v(k)$ enters the system, so the exogenous input $u(k)$ extends the solely (stochastic) autoregressive model, in which $u(k) = 0$.

The optimal predictor for an ARX model is

$$\hat{y}(k) = b_0u(k) + b_1u(k-1) + \ldots + b_mu(k-m) - a_1y(k-1) - \ldots - a_my(k-m)\,. \tag{2.6}$$

Here, m is the order of the model and has to be chosen quite carefully. For simplicity, the order m determines the number of delayed inputs and outputs.[1]

The optimal predictor relation for (2.6) is found by setting $G(q) = \frac{B(q)}{A(q)}$ and $H(q) = \frac{1}{A(q)}$ in the optimal predictor equation (for its derivation see [69])

$$\hat{y}(k) = \frac{G(q)}{H(q)}u(k) + \left(1 - \frac{1}{H(q)}\right)y(k)\,. \tag{2.7}$$

[1] Note that in a more general case, the order of input and output could also be chosen separately.

The ARX model is so popular because the model is linear in the parameters, which leads to a linear optimization problem for parameter estimation. The optimal parameters $\underline{\hat{\theta}}$ for an ARX model of order m are calculated in the following matrix-vector form with $N - m$ (where N is the number of data samples) equations for discrete time steps $k = m + 1, \ldots, N$

$$\underline{\hat{\theta}} = (\underline{X}^T \underline{X})^{-1} \underline{X}^T \underline{y} \,, \tag{2.8}$$

with

$$\underline{\hat{\theta}} = \begin{bmatrix} b_0 & b_1 & \ldots & b_m & a_1 & \ldots & a_m \end{bmatrix}^T \,, \tag{2.9}$$

$$\underline{X} = \begin{bmatrix} u(m+1) & u(m) & \cdots & u(1) & -y(m) & \cdots & -y(1) \\ u(m+2) & u(m+1) & \cdots & u(2) & -y(m+1) & \cdots & -y(2) \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ u(N) & u(N-1) & \cdots & u(N-m) & -y(N-1) & \cdots & -y(N-m) \end{bmatrix} \tag{2.10}$$

and

$$\underline{y} = \begin{bmatrix} y(m+1) & y(m+2) & \ldots & y(N) \end{bmatrix}^T \,. \tag{2.11}$$

The model output is computed by

$$\underline{\hat{y}} = \underline{X} \, \underline{\theta} \,, \tag{2.12}$$

where $\hat{y}(k)$ is in the same form as $y(k)$. The downside of the ARX model is its equation error configuration (see Fig. 2.6). This means that during training (in series-parallel configuration), noisy process outputs are used for parameter estimation, which yields the model assumption that the noise $v(k)$ enters the process before the shared denominator dynamic $\frac{1}{A(q)}$. In turn, this leads to a consistency problem, meaning if the process does not follow this specific noise assumption, all estimated parameters are biased and are not consistent.[2]

OE An output error (OE) model, in contrast, can be obtained by simplifying the general model (2.4) by setting the polynomials $A(q)$, $C(q)$ and $D(q)$ to 1 (see Fig. 2.7)

$$y(k) = \frac{B(q)}{F(q)} u(k) + v(k) \,. \tag{2.13}$$

[2] Non-consistency means that the bias does not tend to zero for an increasing number of samples.

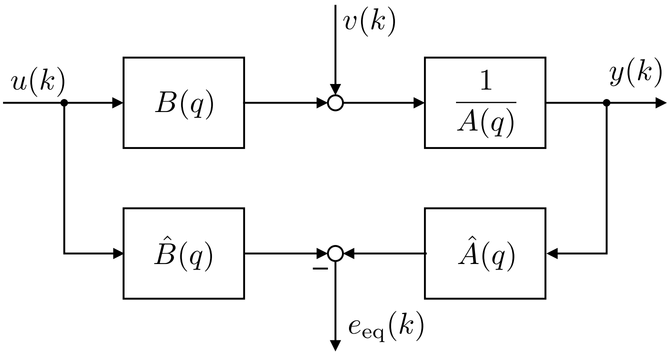


Figure 2.6: Equation error configuration (see also [89])

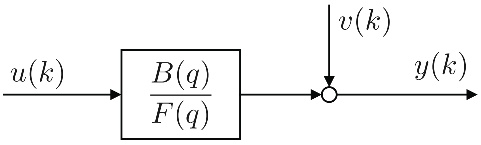


Figure 2.7: OE model structure (see also [89])

The optimal predictor for an OE model is (with $G(q) = \frac{B(q)}{F(q)}$ and $H(q) = 1$)

$$\hat{y}(k) = \frac{B(q)}{F(q)} u(k) \, . \tag{2.14}$$

The optimal predictor, therefore, represents a simulation. Previous outputs are not used because through the white Gaussian noise assumption (no correlation in the noise), no additional information would be available from previous process outputs [89].

The advantage of an OE model is that there is no shared denominator dynamic, which leads to a more realistic noise assumption. However, all OE models are nonlinear in their parameters and, consequently, harder to estimate (nonlinear optimization problem, see Sect. 2.3) than equation error models. The output error configuration for the system and model can be seen in Fig. 2.8.

2.1.1.2 Nonlinear External Dynamics Models

The external dynamics approach for nonlinear modeling has been very popular throughout the years [130, 107, 69, 89]. It is based on the nonlinear input-output model

$$\hat{y}(k) = f(\underline{\varphi}(k)) \, . \tag{2.15}$$

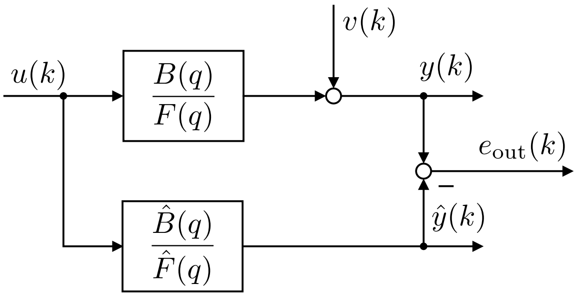


Figure 2.8: Output error configuration (see also [89])

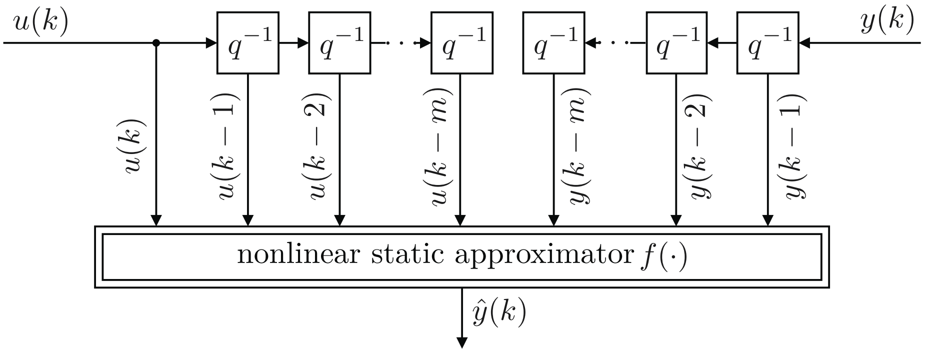


Figure 2.9: Nonlinear external dynamics model (see also [89])

The regression vector $\underline{\varphi}(k)$ contains current and previous inputs and previous system or model outputs. The problem in external dynamics modeling can be clearly separated into selecting an appropriate regression vector $\underline{\varphi}(k)$ and selecting a structure for the nonlinear mapping $f(\cdot)$. Figure 2.9 illustrates this nonlinear external dynamics model, where the regression vector (also called dynamics filter bank) is a *tapped-delay line.*

Choices for the regression vector We will first cover the most common choices for the regression vector and, afterward, deal with the nonlinear static approximator $f(\cdot)$.

NARX Models For the well-known nonlinear autoregressive with exogenous input (NARX) model [18], the regression vector $\underline{\varphi}(k)$ of (2.15) becomes

$$\underline{\varphi}(k) = [u(k)\; u(k-1)\; \dots\; u(k-m)\; y(k-1)\; \dots\; y(k-m)]^T\,, \tag{2.16}$$

where m is the order of the model. This configuration leads to an input space with $2m + 1$ dimensions to the function $f(\cdot)$. This model is the nonlinear extension of the ARX model as the actual *process* outputs $y(k-1), \ldots, y(k-m)$ are fed back. The NARX structure is attractive since the parameters can be estimated under certain circumstances[3] by least squares. However, a NARX model has some significant disadvantages, as the stability of the estimated model is not guaranteed and the one-step prediction error is optimized instead of the simulation error [8]. NARX models are also of interest in recent machine learning research, as it has been shown that temporal convolutional networks (TCNs) can be seen as several NARX models stacked on top of each other [5]. TCNs can therefore be interpreted as deep NARX models. For many layers, often, the outputs are neglected in the regression vector, leading to nonlinear finite impulse response (NFIR) structures (see next but one paragraph). Applications of NARX models with the local linear model tree (LOLIMOT) or hierarchical local model tree (HILOMOT) can be found, for example, in [80, 46, 8].

NOE Models Nonlinear output error (NOE) models are estimated with the regression vector

$$\underline{\varphi}(k) = [u(k)\ u(k-1)\ \cdots\ u(k-m)\ \hat{y}(k-1)\ \cdots\ \hat{y}(k-m)]^T . \tag{2.17}$$

As the *model* outputs $\hat{y}(\cdot)$ are used, nonlinear optimization is the only way to estimate the model. NOE models are advantageous in comparison to NARX and nonlinear autoregressive moving average with exogenous input (NARMAX) models because NOE models yield the optimal simulation error, which is precisely the goal of modeling [89]. It is common to initialize the nonlinear optimization with a NARX model.

NFIR Models When solely current and past inputs are used in the regression vector as

$$\underline{\varphi}(k) = [u(k)\ u(k-1)\ \cdots\ u(k-m)]^T , \tag{2.18}$$

the NFIR model is obtained. The main advantage of this model structure is its inherent stability. The price to pay for the missing feedback is that the dynamic order m has to be chosen very large to describe the process dynamics adequately [89]. This

[3] If the nonlinear approximator $f(\cdot)$ is linearly parameterized.

high order m leads to a high-dimensional input space to the function approximator $f(\cdot)$. Therefore, NFIR models are only found in a few applications [89].

Choices of Nonlinear Static Approximator Now we will consider the nonlinear mapping $f(\cdot)$. Since $\underline{\varphi}(k)$ is now set, principally, any static function approximator may be used.

Basis Function Formulation A common choice is the basis function formulation

$$\hat{y}(k) = f(\underline{\varphi}(k)) = \sum_{j=1}^{n_m} \theta_j \Phi_j \left(\underline{\varphi}(k), \underline{\theta}_j^{[nl]} \right) . \tag{2.19}$$

A detailed explanation of the basis function formulation and neural networks is given in Sect. 2.2. The output $\hat{y}(k)$ is calculated as a weighted sum of n_m basis functions $\Phi_j(\cdot)$. If $f(\cdot)$ is linear in its parameters, the estimation of a NARX or NFIR model can be solved with least squares which is a huge advantage for those model structures. For linearity in the parameters, structures like polynomials, radial basis function (RBF) networks, or local model networks (LMNs) are used. If a function $f(\cdot)$ is chosen that is nonlinear in the parameters (which is inevitably the case for NOE structures), this advantage disappears and the model parameters need to be estimated by means of nonlinear optimization.

Gaussian Processes Instead of choosing a parametric model for $f(\cdot)$, non-parametric models can also be chosen. Here, Gaussian processes (GPs) play the most important role [103]. The key idea is to place a so-called kernel on every data point in the input space, which measures the closeness/similarity (usually in the form of a norm) of one data point to all other data points in the input space. Then, the reasoning is that if two data points are "close" in the input space, their output values should also be somewhat similar (given the assumption of a smooth process). This is done in a probabilistic framework and means that not a point estimate $\hat{y}(k) = f(\underline{\varphi}(k))$ is calculated, but a complete probability density function (PDF) is modeled. The PDFs are Gaussians, hence the name Gaussian processes.

First, a prior over all admissible functions $f(\cdot)$ is assumed, representing the modeler's believes about the mapping [61], which usually incorporates some smoothness assumptions. The prior is then updated by the information contained in the measured data (the likelihood) to form a posterior distribution. Modeling PDFs has the advantage to quantify uncertainty that deterministic models are not capable of [61]. A thorough account on GPs in general can be found in [103]. For a description of NARX models with GPs, the reader is referred to [61].

2.1.2 Internal Dynamics Models

Models with internal dynamics are based on the extension of external dynamics models with internal memory [89]. This internal memory is represented by the so-called state vector $\underline{\hat{x}}(k)$. The space spanned by the state vector is called the state space.

As for the external dynamics models, we will first cover the linear basics and then turn to different nonlinear internal dynamics models.

2.1.2.1 Linear Internal Dynamics Models

An n_x-th order SISO discrete-time linear state space model is represented by

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \underline{A}\,\underline{\hat{x}}(k) + \underline{b}\,u(k), \quad \underline{\hat{x}}(0) = \underline{\hat{x}}_0 \\ \hat{y}(k) &= \underline{c}^T\,\underline{\hat{x}}(k) + d\,u(k)\,. \end{aligned} \tag{2.20}$$

The first equation is called the state (update) equation, whereas the second is called the output equation. The state vector $\underline{\hat{x}}(k) = [\hat{x}_1(k)\,\hat{x}_2(k)\,\ldots\,\hat{x}_{n_x}(k)]^T \in \mathbb{R}^{n_x}$ contains n_x state variables at the discrete time step k. The input and output are denoted by $u(k)$ and $\hat{y}(k)$, respectively. The parameters are stored in the matrix, vectors, and scalar $\underline{A} \in \mathbb{R}^{n_x \times n_x}$, $\underline{b} \in \mathbb{R}^{n_x}$, $\underline{c}^T \in \mathbb{R}^{1 \times n_x}$, and feed-through $d \in \mathbb{R}$.

In Fig. 2.10, the block diagram of the linear state space model is depicted. Matrix $\underline{A}$ maps a weighted sum of current states and $\underline{b}$ the current input to the updated state vector $\underline{\hat{x}}(k+1)$ at time step $k+1$. This means that each updated state variable is a linear combination of previous state variables and the previous input. All model parameters are stored in a parameter matrix

$$\underline{\Theta} = \begin{bmatrix} \underline{A} & \underline{b} \\ \underline{c}^T & d \end{bmatrix}, \tag{2.21}$$

or in a parameter vector

$$\underline{\theta} = \mathrm{vec}(\underline{\Theta})\,, \tag{2.22}$$

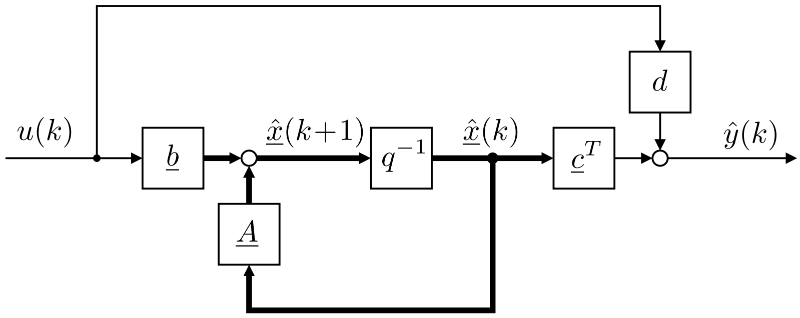


Figure 2.10: Block diagram of a SISO linear state space model (see also [72]). Thin and bold lines are used for scalar and vectorial quantities, respectively.

where the vec(·) operator stacks all matrix entries column-wise.

Similarity Transformation of Linear State Space Models State space models possess n_x^2 redundant parameters and are therefore a non-unique representation. Consider the transformed state vector

$$\tilde{\underline{x}}(k) = \underline{T}^{-1}\hat{\underline{x}}(k)\,, \tag{2.23}$$

where $\underline{T} \in \mathbb{R}^{n_x \times n_x}$ is an arbitrary non-singular matrix. This transformation leads to an infinite number of model parameterizations, which all cause the model to have the same input-output behavior. The class of all input-output-equivalent state space models is described by [152]

$$S_\Theta(\underline{T}) = \begin{bmatrix} \tilde{\underline{A}} & \tilde{\underline{b}} \\ \tilde{\underline{c}}^T & \tilde{d} \end{bmatrix} = \begin{bmatrix} \underline{T}^{-1}\underline{A}\underline{T} & \underline{T}^{-1}\underline{b} \\ \underline{c}^T\underline{T} & d \end{bmatrix} = \begin{bmatrix} \underline{T}^{-1} & \underline{0} \\ \underline{0} & 1 \end{bmatrix} \begin{bmatrix} \underline{A} & \underline{b} \\ \underline{c}^T & d \end{bmatrix} \begin{bmatrix} \underline{T} & \underline{0} \\ \underline{0} & 1 \end{bmatrix} . \tag{2.24}$$

Conversion of Linear Internal Dynamics Model to External Dynamics Model Every linear state space model can be converted into a linear external dynamics model with the relation

$$G(z) = \underline{c}^T(z\underline{I} - \underline{A})^{-1}\underline{b} + d\,, \tag{2.25}$$

where z is the time-shift operator q in the frequency domain [152]. Vice versa, every linear external dynamics model can be converted into the state space representation. However, as shown previously, there is an infinite number of state space representations for one input-output model due to the non-uniqueness of the state space representation.

2.1.2.2 Best Linear Approximation

The best linear approximation (BLA) framework [29, 30, 101] will be used as starting point for the local model state space network (LMSSN). An overview of practical applications of the BLA and its usefulness is given in [64, 118]. As a criterion for the BLA, [116] defined the best linear transfer function $G_{\mathrm{BLA}}(q)$ to be the linear model that minimizes a loss function between process and linear model output in a least squares sense as

$$G_{\mathrm{BLA}}(q) = \underset{G(q)}{\arg\min}\, \mathbb{E}\left\{ |y(k) - G(q)u(k)|^2 \right\} , \tag{2.26}$$

where $\mathbb{E}\{\cdot\}$ denotes the expected value operator. The BLA is obtained by the following three steps.

1. Estimation of the Nonparametric BLA A system can be modeled as the sum of a nonparametric linear system $\hat{G}_{\mathrm{BLA}}(j\omega_n)$ and a transfer function $Y_S(j\omega_n)$ for the noise source $y_s(k)$ (see Fig. 2.11). The noise source represents the part of the output $y(k)$ that the BLA cannot capture, i.e., noise and the nonlinear component combined. Hence, the output can be written in the frequency domain as

$$Y(j\omega_n) = \hat{G}_{\mathrm{BLA}}(j\omega_n)U(j\omega_n) + Y_S(j\omega_n) \tag{2.27}$$

for the discrete frequencies ω_n. The nonparametric BLA is then calculated (as shown in [101]) by

$$\hat{G}_{\mathrm{BLA}}(j\omega_n) = \frac{S_{yu}(j\omega_n)}{S_{uu}(j\omega_n)}, \tag{2.28}$$

where $S_{yu}(j\omega_n)$ is the cross-power spectrum between the output and the input and $S_{uu}(j\omega_n)$ is the auto-power spectrum of the input [96]. Equation (2.28) is obtained by evaluating the Fourier transform of the Wiener-Hopf equation which follows from (2.26), see [32].

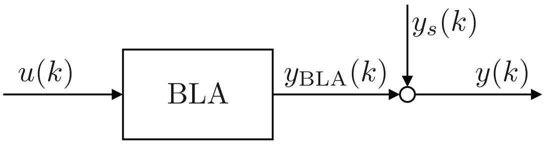


Figure 2.11: Representation of BLA and noise source (see also [96])

For periodic signals, (2.28) reduces to the average of the measured frequency response functions (FRFs) for M experiments as

$$\hat{G}_{\mathrm{BLA}}(j\omega_n) = \frac{1}{M}\sum_{m=1}^{M} \hat{G}^{[m]}(j\omega_n)\,. \tag{2.29}$$

Here, for every experiment m, the FRF estimate $\hat{G}^{[m]}(j\omega_n)$ is defined as

$$\hat{G}^{[m]}(j\omega_n) = \frac{\hat{Y}^{[m]}(j\omega_n)}{\hat{U}^{[m]}(j\omega_n)}\,, \tag{2.30}$$

with

$$\hat{U}^{[m]}(j\omega_n) = \frac{1}{P}\sum_{p=1}^{P} U^{[m,p]}(j\omega_n) \tag{2.31}$$

$$\hat{Y}^{[m]}(j\omega_n) = \frac{1}{P}\sum_{p=1}^{P} Y^{[m,p]}(j\omega_n)\,, \tag{2.32}$$

where $U^{[m,p]}(j\omega_n)$ and $Y^{[m,p]}(j\omega_n)$ denote the discrete Fourier transforms (DFTs) of the p-th period of experiment m in a periodic excitation setup. The total number of periods is denoted by P.

For non-periodic input signals, one can estimate the BLA with the auto-power and cross-power spectra. In this case, leakage errors will be present when calculating the DFTs [96]. First, the input signal is split into M blocks. A Hanning or Diff window can be applied to the signals to reduce the leakage effect. Next, the DFT spectra of each block are estimated. From that, the sample cross-power spectrum between the output and input $\hat{S}_{YU}(j\omega_n)$ and the auto-power spectrum of the input $\hat{S}_{UU}(j\omega_n)$ are calculated using Welch's method [142]:

$$\hat{S}_{YU}(j\omega_n) = \frac{1}{M}\sum_{m=1}^{M} Y^{[m]}(j\omega_n)U^{[m]H}(j\omega_n)\,, \tag{2.33}$$

$$\hat{S}_{UU}(j\omega_n) = \frac{1}{M}\sum_{m=1}^{M} U^{[m]}(j\omega_n)U^{[m]H}(j\omega_n)\,. \tag{2.34}$$

Here, $(\cdot)^H$ denotes the conjugate transpose of a matrix. The nonparametric BLA is then given by

$$\hat{G}_{\mathrm{BLA}}(j\omega_n) = \frac{\hat{S}_{YU}(j\omega_n)}{\hat{S}_{UU}(j\omega_n)}\,. \tag{2.35}$$

2. Estimation of Linear Parametric Model based on BLA A parametric linear state space model is estimated from the nonparametric BLA using frequency domain subspace identification methods [78]. The frequency weighting by means of the weighted least squares loss function is stated by [100]

$$I_{\mathrm{WLS}} = \sum_{n=1}^{F} \varepsilon^{H}(j\omega_n)\hat{C}_G^{-1}(j\omega_n)\varepsilon(j\omega_n)\,. \tag{2.36}$$

Here, F is the number of discrete frequencies, $\varepsilon(j\omega_n)$ is the difference of the nonparametric and parametric BLA at the discrete frequency ω_n, and the sample covariance $\hat{C}_G(j\omega_n)$ is given by [98]

$$\hat{C}_G(j\omega_n) = \frac{1}{M(M-1)} \sum_{m=1}^{M} |\hat{G}^{[m]}(j\omega_n) - \hat{G}_{\mathrm{BLA}}(j\omega_n)|^2\,. \tag{2.37}$$

The frequency weighting is only used when multiple realizations of the experiment are available ($M > 1$). In this way, the initial parameters $\underline{A}$, $\underline{b}$, $\underline{c}^T$, and d are estimated for the parametric BLA.

3. Nonlinear Optimization of the Linear Parametric Model The initial parameters are tuned by nonlinear optimization. The Levenberg-Marquardt algorithm (see Sect. 2.3.2) is employed [98] to accomplish this. If multiple realizations are available, the nonlinear optimization is also performed with a frequency weighting of the sample covariance matrix $\hat{C}_G(j\omega_n)$. For an in-depth analysis of the BLA, especially for initialization of nonlinear models, refer to [96].

Case Study: Initialization Example on Silverbox Benchmark For demonstration purposes, the second-order BLA for the Silverbox benchmark (Sect. 5.4) is shown in Fig. 2.12. Here, the nonparametric BLA (blue dots), the fitted parametric second-order BLA (yellow line), and the error at the discrete frequencies f (red dots) are shown. Only odd frequencies were excited in a frequency range from 0 to 200 Hz (in total 1342 frequency lines) and therefore the fitting was only done for odd frequencies, since fitting on even frequencies would only contribute noise (no excitation). The frequency weighting is used as the training input consists of $M = 10$ realizations of a random odd multisine signal. One can see that the parametric model fits the nonparametric BLA well and the error between those models is acceptably low.

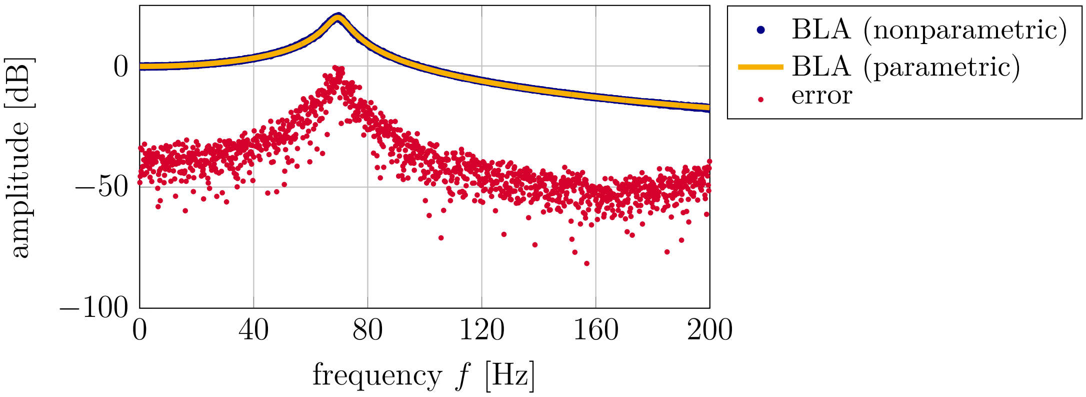


Figure 2.12: Demonstration of the BLA on Silverbox benchmark ($M = 10$ experiments)

2.1.2.3 Nonlinear Internal Dynamics Models

Now we will take a closer look at nonlinear internal dynamics models. As for the linear case, we write a nonlinear internal dynamics models in state space representation (SISO case) as

$$\begin{aligned}\underline{\hat{x}}(k+1) &= \underline{h}(\underline{\hat{x}}(k), u(k)) \\ \hat{y}(k) &= g(\underline{\hat{x}}(k), u(k))\,.\end{aligned} \tag{2.38}$$

The functions $\underline{h}(\cdot)$ and $g(\cdot)$ describe a general functional dependence of $\underline{\hat{x}}(k)$ and $u(k)$ in the state and output equation.

The identification task is to determine the functions $\underline{h}(\cdot)$ for the n_x state variable updates and the function $g(\cdot)$. Due to the internal feedback, an optimization problem formulated with a nonlinear state space representation leads to a nonlinear optimization problem.

Similarity Transformation of Nonlinear State Space Models Just like for linear state space models, also nonlinear models can be transformed to yield an infinite number of models with the same input-output behavior [98]. The similarity transform $\underline{\tilde{x}}(k) = \underline{T}^{-1}\underline{\hat{x}}(k)$ with arbitrary nonsingular square matrix $\underline{T}$ yields in the nonlinear case

$$\begin{aligned}\underline{\tilde{x}}(k+1) &= \underline{T}^{-1}\underline{h}(\underline{T}\,\underline{\tilde{x}}(k), u(k)) = \underline{\tilde{h}}(\underline{\tilde{x}}(k), u(k)) \\ \hat{y}(k) &= g(\underline{T}\,\underline{\tilde{x}}(k), u(k)) = \tilde{g}(\underline{\tilde{x}}(k), u(k))\,.\end{aligned} \tag{2.39}$$

The transformation by $\underline{T}$ shows that there are n_x^2 redundant parameters. This property can be extended by an offset $\underline{t}$ which allows not just to scale but also to shift the state vector as $\underline{\tilde{x}}(k) = \underline{T}^{-1}(\underline{\hat{x}}(k) - \underline{t})$, which leads to

$$\begin{aligned} \underline{\tilde{x}}(k+1) &= \underline{T}^{-1}\left(\underline{h}(\underline{T}\,\underline{\tilde{x}}(k) + \underline{t}, u(k)) - \underline{t}\right) = \underline{\tilde{h}}(\underline{\tilde{x}}(k), u(k)) \\ \hat{y}(k) &= g(\underline{T}\,\underline{\tilde{x}}(k) + \underline{t}, u(k)) = \tilde{g}(\underline{\tilde{x}}(k), u(k))\,. \end{aligned} \tag{2.40}$$

In the affine case, also the initial condition has to be transformed according to

$$\underline{\tilde{x}}_0 = \underline{T}^{-1}(\underline{\hat{x}}_0 - \underline{t})\,. \tag{2.41}$$

The affine transformation will be used for the LMSSN to shift the state trajectory in the desired operating regime (see Sect. 3.3.3 and Appx. B.2 for more detail). Note that there are now $n_x^2 + n_x$ redundant parameters in the affine transformation case.

PNLSS The polynomial nonlinear state space model (PNLSS) was developed by Paduart (2008) [96]. It extends the linear state space model (2.20) by two additional terms which add higher-order polynomials to the state and output equation. PNLSS seems to be one of the most promising deterministic approaches that has been developed in recent years in nonlinear system identification. Further improvements have been suggested for the PNLSS, e.g., the use of decoupling strategies to cope with the curse of dimensionality for polynomials [34, 35, 33, 25, 115].

For SISO systems, the PNLSS model is depicted in Fig. 2.13. It can be written as

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \underline{A}\,\underline{\hat{x}}(k) + \underline{b}\,u(k) + \underline{E}\underline{\zeta}(\underline{\hat{x}}(k), u(k)) \\ \hat{y}(k) &= \underline{c}^T\underline{\hat{x}}(k) + d\,u(k) + \underline{f}^T\underline{\eta}(\underline{\hat{x}}(k), u(k))\,. \end{aligned} \tag{2.42}$$

The vectors $\underline{\zeta}(\underline{\hat{x}}(k), u(k))$ and $\underline{\eta}(\underline{\hat{x}}(k), u(k))$ contain nonlinear monomials in $\underline{\hat{x}}(k)$ and $u(k)$ of degree two up to a chosen degree p. The coefficients associated with these nonlinear terms are the matrix $\underline{E}$ and the vector $\underline{f}^T$. Note that the monomials of degree one are included in the linear part of the PNLSS model. For a second-order SISO model, the monomials of degree $p = 2$ in the state equations are, for example[4],

$$\underline{\zeta}(\underline{\hat{x}}(k), u(k)) = [\hat{x}_1^2 \;\; \hat{x}_1\hat{x}_2 \;\; \hat{x}_1 u \;\; \hat{x}_2^2 \;\; \hat{x}_2 u \;\; u^2]^T\,. \tag{2.43}$$

[4] The argument k of $\underline{\hat{x}}(k)$ and $u(k)$ is left out for brevity.

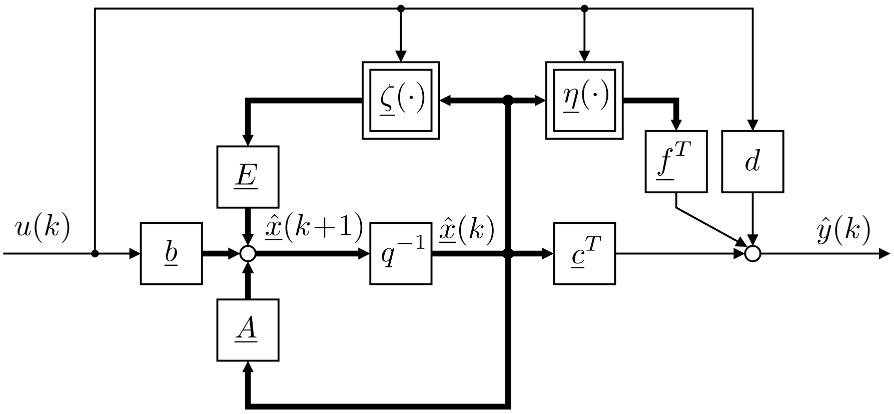


Figure 2.13: Block diagram of polynomial nonlinear state space model

The identification of the PNLSS model is as follows [98]:

1. Find the BLA [101] of the system under test as the nonparametric transfer function $\hat{G}_{\text{BLA}}$.

2. Based on the nonparametric BLA, calculate the parametric BLA G_{BLA} and therefore the initial estimates of the linear parameters (all entries in $\underline{A}, \underline{b}, \underline{c}^T$ and d), using a linear subspace system identification procedure [100] followed by a nonlinear optimization of the parameters.

3. Tune all parameters $\underline{A}, \underline{b}, \underline{c}^T, d, \underline{E}$ and $\underline{f}^T$ by running a Levenberg-Marquardt optimization. Matrices $\underline{E}$ and $\underline{f}^T$ are initially set to $\underline{0}$ [98].

The whole identification procedure can be found in [96] or [98]. The strengths of the PNLSS model are its great flexibility and adaptability. The estimates of the BLA can be used to initialize the system, which means that the PNLSS model is performing equally well as the best linear model or better (on training data).

On the other hand, the PNLSS suffers from stability issues and explosive behavior in extrapolation – in regions where they were not estimated [96, 35]. Those problems are even more pronounced for a high order p, making it hard to estimate satisfactory models if operated close to the training data boundaries and in extrapolation. Therefore, usually low polynomial degrees of $p = 2$ or $p = 3$ are chosen. Complex PNLSS models do easily get unstable in extrapolation, as shown in [125, 93]. For the decoupled PNLSS [34, 35], the authors point out that the amplitudes of the input signal for testing has to be chosen smaller than for training, as otherwise extrapolation problems occur. This behavior constitutes a significant limitation to the usefulness of those methods.

Deep Probabilistic State Space Networks There is much research done in probabilistic state space models [114]. Recent research has shown the usefulness of combining recurrent neural networks (RNNs) (which can be transformed into state space models, see Sect. 4.1) and variational autoencoders [60], especially as those have the advantage of quantifying uncertainty [39]. The six different methods implemented in [39] generally perform well but are interestingly outperformed by the BLA (a linear model) on a Wiener-Hammerstein with process noise benchmark [121]. The better performance of the BLA might be caused by the chosen double-sided saturation nonlinearity for the benchmark, which might only exhibit slightly nonlinear behavior.

Piecewise Affine State Space Models A piecewise affine (PWA) state space model [38] is described by

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \underline{o}_j + \underline{A}_j\, \underline{\hat{x}}(k) + \underline{b}_j u(k) \\ y(k) &= p_j + \underline{c}_j^T \underline{\hat{x}}(k) + d_j u(k) \\ \text{if} &\begin{bmatrix} \underline{\hat{x}}(k) \\ u(k) \end{bmatrix} \in \mathcal{X}_j, \quad j = 1, \ldots, n_m\,, \end{aligned} \tag{2.44}$$

where $\underline{o}_j$ and p_j are the offsets for state and output equation and n_m is the number of affine models. Depending on the operating point in the joint input space $[\underline{\hat{x}}^T(k)\ u(k)]^T$, a single affine state space model governs the process dynamics. This means that all model parameters in state and output equation switch simultaneously depending on the operating point, leading instantaneously to different model behavior. This may be preferable for hybrid systems[5], but is usually not desired for processes, where a smooth and continuous function behavior is expected.

Multiple Affine State Space Models Multiple model approaches [86] can alleviate the "hard switching" problem of PWA state space models. Proposed model structures in state space formulation are given by

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \sum_{j=1}^{n_m} \Big(\underline{o}_j + \underline{A}_j\, \underline{\hat{x}}(k) + \underline{b}_j u(k)\Big)\, \varPhi_j(k) \\ \hat{y}(k) &= \sum_{j=1}^{n_m} \Big(p_j + \underline{c}_j^T\, \underline{\hat{x}}(k) + d_j u(k)\Big)\, \varPhi_j(k)\,. \end{aligned} \tag{2.45}$$

[5] Those are processes that combine continuous (described by differential equations) and discrete (described by finite state logic) dynamic behavior [10].

Here, a number of n_m affine state space models is weighted to form an overall blended model. It has been shown in [58] that this model is a universal approximator to the system (2.38), as an arbitrarily small modeling error can be achieved for a sufficiently large number of affine models n_m [139]. Like in the PWA case, all parameters in the state and output equation are treated as a whole. Therefore, one can think of this strategy to blend affine state space models as a whole and not just distinct parts of them, which will be further elaborated on in Chap. 3. This will play a crucial role in the LMSSN, where only dedicated parts of state space models can be blended over. Model structures as in (2.45) are in detail elaborated on in [139] and [152].

Neural Network State Space Models In the nonlinear state space model (2.38), the functions $\underline{h}(\cdot)$ and $g(\cdot)$ can be approximated by neural networks [130]. For example, single layer perceptrons have been studied in the 1990s [76, 150], while mostly deep network architectures are investigated today [123].

2.1.3 Comparison of Internal versus External Dynamics Realization

Internal (state space) models and external (input-output) models are two different modeling approaches with various advantages and disadvantages. A non-comprehensive overview shall be given in the following.

- As argued by [106], state space models can describe a wider class of dynamical systems than input-output models. It is always possible to write a nonlinear input-output model in a state space representation, but not all state space models can be formulated into input-output representation.
- Additionally argued by [106], even if an input-output representation exists of the state space model, the state space model may require a lower order.
  When comparing an input-output and state space model both of order n_x, a state space model has n_x^2 more parameters than an input-output model (due to the non-uniqueness of the state space model). Those n_x^2 parameters are redundant, which means that the effective number of parameters is the same for the input-output and state space model. If now the same representational capacity is achieved with a lower order state space model, then this might also lead to a smaller effective number of parameters for state space models.

- State space models can easily be extended to the MIMO case, which is more cumbersome for input-output models.

- The number of regressors and, therefore, the dimensionality of the input space is usually smaller in state space models than in input-output models. A n_x-th order SISO state space model possesses $n_x + 1$ regressors $\underline{X} = [\hat{x}_1(k) \ \dots \ \hat{x}_{n_x}(k) \ u(k)]$, while an input-output model of m-th order requires $2m{+}1$ regressors $\underline{X} = [u(k) \ u(k{-}1) \ \dots \ u(k{-}m) \ y(k{-}1) \ \dots \ y(k{-}m)]$. This is especially important for nonlinear systems as the model complexity, the computational demand, and the amount of data needed for the function approximation increases significantly with the input space dimensionality (curse of dimensionality) [89].

- Modern control methods often require state space models instead of input-output models [139].

- Training of input-output models can be done in a non-recurrent manner with equation-error approaches. State space models instead are recurrent [91, 92], making optimization a lot more involved.

- State space models are non-unique since an infinite number of similarity transformations can be applied to the state space model, which all lead to the same input-output behavior of the model (see Sect. 2.1.2.3).

- State space models are more abstract than input-output models.

2.1.4 Block-Oriented Models

As pointed out in [110], often pure black-box models are used for processes, where partial insights have been gained by first principles. The belief might be that an interpretable model could not possibly be as accurate as a black box for a complex dataset. Black-box models do have their justification, but if a more interpretable, more parsimonious model can be employed, it should be. Throughout the past 30 years, much attention has been given to the topic of block-oriented nonlinear system identification [87, 41, 122]. Here, nonlinear systems are modeled only by linear time-invariant (LTI) dynamic subsystems and static nonlinearities, making those models readily understandable. Through different choices of interconnections, a multitude of nonlinear models can be obtained.

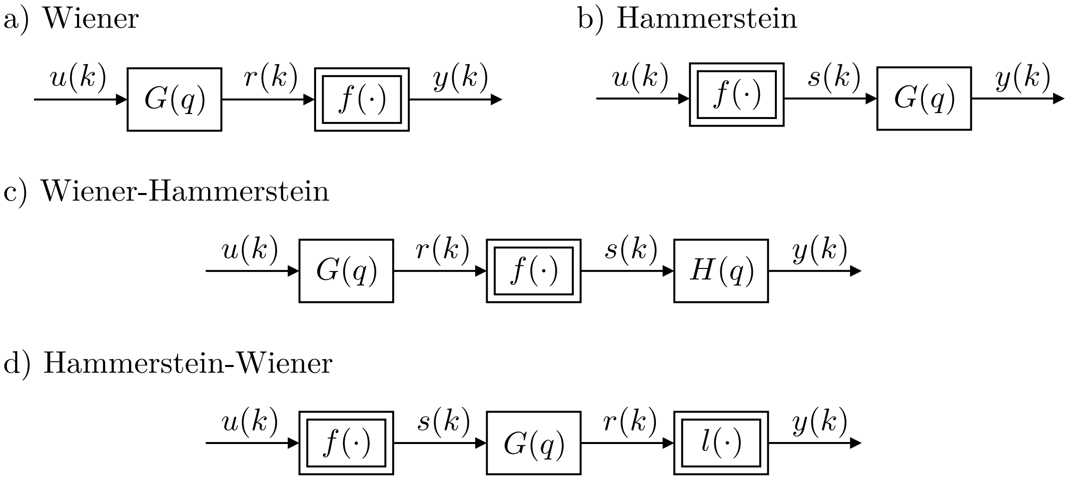


Figure 2.14: Considered single branch block-oriented structures. The LTI blocks ($G(q)$, $H(q)$) and static nonlinearity blocks ($f(\cdot)$, $l(\cdot)$) are differently combined to form the different models a) - d). Signals that exit an LTI block are labeled as $r(k)$ and signals exiting static nonlinearities are labeled $s(k)$.

Considering only single branch structures, the most popular and most studied block-oriented structures are shown in Fig. 2.14. A Wiener system (Fig. 2.14 a) is a series connection of an LTI block followed by a static nonlinearity. It is extended to a Wiener-Hammerstein system (Fig. 2.14 c) by series-connecting another LTI block after the static nonlinearity. A Hammerstein system (Fig. 2.14 b) is "the opposite" of a Wiener system. First, the signal is passed through a static nonlinearity and then passed through an LTI block. It can likewise be extended to a Hammerstein-Wiener system (Fig. 2.14 d) by connecting another static nonlinearity to the output of the LTI block of the Hammerstein system. A non-exhaustive overview of different identification methods for the different model structures is given in [122]. Some similarities of the LMSSN to block-oriented models are shown in Sect. 3.5.2.

2.2 Neural Networks

The simplest static model is a *linear* static model. The output $\hat{y}$ is, in this case, a linear combination of the inputs. This can be written as the scalar product of an input vector $\underline{u}$ with a parameter vector $\underline{\theta}$ as

$$\hat{y} = \underline{\theta}^T \underline{u} \,. \tag{2.46}$$

This equation simply resembles a straight line through the origin of the coordinate system in the SISO case or a (hyper-)plane for the MISO case. Parameter vector $\underline{\theta}$ denotes the slopes in all n_p input dimensions (entries in vector $\underline{u}$).

In contrast, a *nonlinear* static model can generally be described by

$$\hat{y} = f(\underline{u}) \,. \tag{2.47}$$

The parameterization and estimation of function $f(\cdot)$ can be understood as system identification for static applications. Since the class of nonlinear models is extensive, many different parametrization approaches exist for $f(\cdot)$. Among those are more classical RBF approaches [11], multilayer perceptron networks [108], or more recently used deep neural network structures [44]. The basic concepts will be explained in the subsequent sections.

2.2.1 Basis Function Formulation

Almost all possible realizations of function $f(\cdot)$ of practical interest can be written in a basis function formulation [89]

$$\hat{y} = \sum_{j=1}^{n_m} \theta_j \Phi_j \left(\underline{u}, \underline{\theta}_j^{[nl]} \right) \,. \tag{2.48}$$

The output $\hat{y}$ is calculated as a weighted sum of n_m basis functions $\Phi_j(\cdot)$. The basis functions are weighted with the linear parameters θ_j and depend on the input vector $\underline{u}$ and a set of nonlinear parameters $\underline{\theta}_j^{[nl]}$. Figure 2.15 illustrates such a general configuration.

All $\Phi_j(\cdot)$ can be, generally speaking, different basis functions. However, if all $\Phi_j(\cdot)$ have the same structure and only differ in $\underline{\theta}_j^{[nl]}$, the basis function network is called a *neural network*. In Fig. 2.15, a neural network is shown with three so-called layers: the input layer, a hidden layer, and the output layer. The representational capabilities of a neural network can be increased in two ways. On the one hand, the number of neurons (or nodes) n_m in the hidden layer can be increased. On the other hand, the number of layers can be increased, leading to deep neural networks.

The general idea behind neural networks has been around for more than 75 years [77]. They have been employed fruitfully in system identification throughout the years

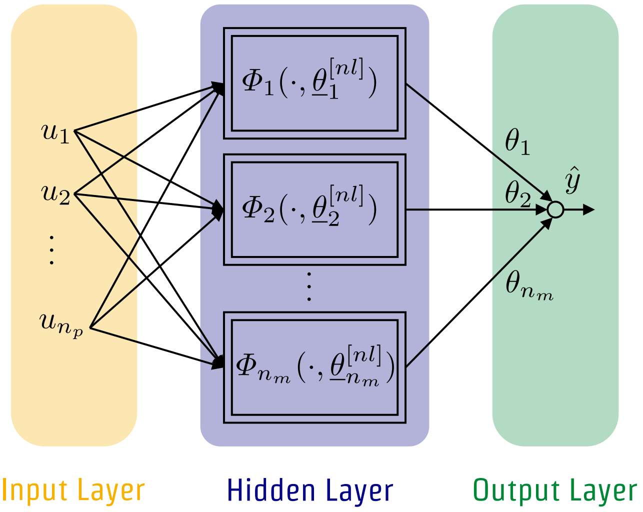

u_1
u_2
u_{n_p}
$\Phi_1(\cdot,\underline{\theta}_1^{[nl]})$
$\Phi_2(\cdot,\underline{\theta}_2^{[nl]})$
$\Phi_{n_m}(\cdot,\underline{\theta}_{n_m}^{[nl]})$
θ_1
θ_2
θ_{n_m}
$\hat{y}$
Input Layer
Hidden Layer
Output Layer

Figure 2.15: Network of basis functions. If all basis functions are of the same type, this architecture is called a neural network.

[87, 17, 130] and still constitute a popular choice for nonlinear system identification [137, 75, 24, 63, 102, 5].

One of their most important properties is that they are universal approximators [57]. This means that they can approximate any arbitrary continuous function when the number of neurons is large enough, even if all neurons are in only one hidden layer. This property makes this class of models well suited for a large class of problems in nonlinear system theory [87]. Some basic neural network structures are illustrated in Appx. A.1.

2.2.2 Validity and Activation Functions

It is frequently desirable to have a *local* interpretability of each neuron. Therefore, it might be helpful to consider different types of nonlinear functions that possess *locality*. Thus, we will investigate normalized radial basis functions (NRBFs) and hierarchical sigmoid functions in detail, as those will be employed for the LMSSN.

Normalized Radial Basis Functions A special case for a neural network is the NRBF network with Gaussian basis functions. In this case, the nonlinear parameters $\underline{\theta}_j^{[nl]}$ of $\Phi_j(\cdot)$ become centers $\underline{\mu}_j$ and standard deviations $\underline{\sigma}_j$ of RBFs. The NRBFs are calculated by

$$\Phi_j\left(\underline{u},\underline{\mu}_j,\underline{\sigma}_j\right)=\frac{\Psi_j(\underline{u},\underline{\mu}_j,\underline{\sigma}_j)}{\sum_{s=1}^{n_m}\Psi_s(\underline{u},\underline{\mu}_s,\underline{\sigma}_s)} \tag{2.49}$$

with the RBFs (which in the fuzzy logic context are also called membership functions)

$$\Psi_j\left(\underline{u}, \underline{\mu}_j, \underline{\sigma}_j\right) = \exp\left(-\frac{1}{2}\left(\frac{(u_1 - \mu_{j1})^2}{\sigma_{j1}^2} + \cdots + \frac{(u_{n_m} - \mu_{jn_m})^2}{\sigma_{jn_m}^2}\right)\right). \tag{2.50}$$

The NRBFs have the special property of summing up at every point in the input space to 1. This property is called *partition of unity* and it holds that

$$\sum_{j=1}^{n_m} \Phi_j(\cdot) = 1\,. \tag{2.51}$$

An NRBF can be interpreted on the one hand the same way as in the basis function formulation. A certain basis function is weighted with a linear parameter. On the other hand, the typical interpretation is to understand the basis function as a validity function. The validity function (or activation function) expresses in which areas of the input space, which weight (which linear parameter) is valid (or active) to what degree. The input space is thereby decomposed into smaller operating regimes which hopefully facilitates decent approximations of the nonlinear system. This interpretation is common because of the partition of unity which can also be understood as some kind of proportion to which a weight is active in a certain region of the input space. This second interpretation will also be used for all further discussion.

The question arises, how the nonlinear parameters, namely centers and standard deviations of the RBFs, are determined. This task can be solved by different means such as grid partitioning, input space clustering, heuristic construction algorithms, or others. One concept, which falls into the category of heuristic algorithms, is a tree-construction algorithm that will be used in this thesis and is explained in more detail in Sect. 3.2.1.

NRBF networks suffer from two problems. On the one hand, since the RBFs are exponentially decaying their normalization becomes increasingly harder with increasing distance to their center coordinates. To avoid a division-by-zero, usually, the smallest value that a chosen numeric data type can represent is added to the denominator. This leads to imprecise calculations of the validity functions far from the center coordinates. The second problem is reactivation [128]. This occurs if two RBFs have different standard deviations. Both problems are discussed in more detail in Sect. 3.6.1.

Hierarchical Sigmoid Functions These two problems for NRBFs can be overcome by choosing alternative validity functions. For example, hierarchical sigmoid functions do not suffer from the previously stated problems.

The general idea is to construct validity functions Φ_j by the multiplication of splitting functions Ψ_j through a hierarchy [89]. Each splitting function is a (multidimensional) sigmoid function

$$\Psi_j(\underline{u}) = \frac{1}{1 + \exp(-z_j)} \quad \text{with} \quad z_j = \kappa \cdot (v_{j0} + v_{j1} u_1 + \ldots + v_{jn_p} u_{n_p}) \,. \tag{2.52}$$

The splitting weights $v_{j1}, \ldots, v_{jn_p}$ are stored in the row vector $\underline{v}_j^T = [v_{j1}, \ldots, v_{jn_p}]$. They determine the direction of the nonlinearity and their ratio with respect to v_{j0} determines the distance from the origin of the coordinate system [47]. Parameter κ is redundant but can be used to adjust the steepness of the sigmoid function conveniently. The goal is to obtain similar behavior to the NRBF case: A neuron is active in a particular local region and inactive everywhere else while guaranteeing that the sum of all validity functions sums up to one.

Figure 2.16 contrasts validity functions constructed by RBFs (left side) and sigmoid

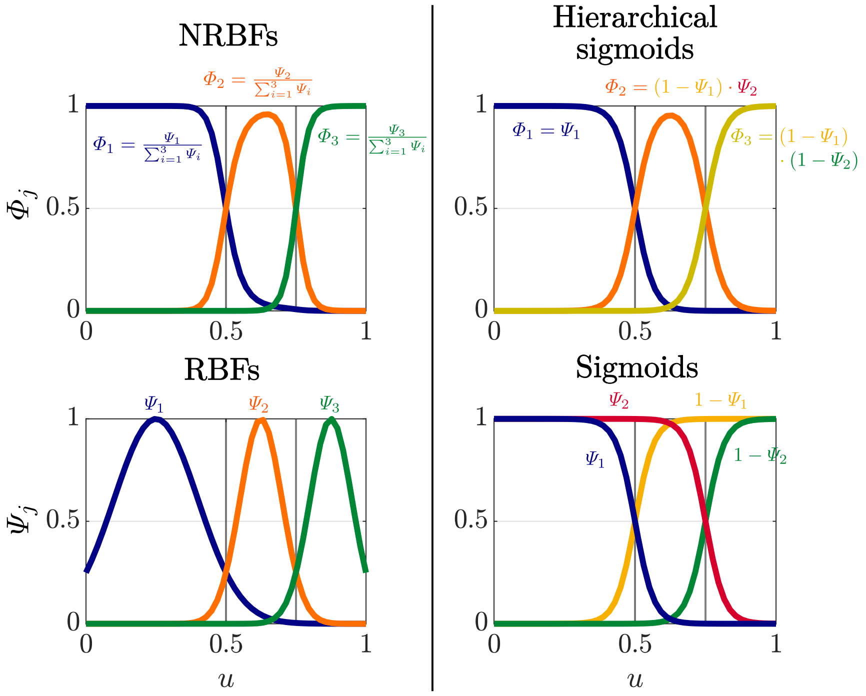


Figure 2.16: Construction of qualitatively equivalent NRBF validity functions (top left) and hierarchical sigmoid validity functions (top right) constructed by RBFs (bottom left) and sigmoids (bottom right), respectively.

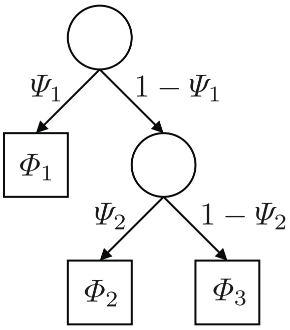


Figure 2.17: Hierarchical sigmoids of the given example. The validity functions (leaves of the tree) are constructed by multiplying all sigmoids leading up to each leaf.

functions (right side). For example, qualitatively equivalent results can be achieved in a given region (in this case for $u \in [0, 1]$) for three validity functions. The NRBFs validity functions are constructed by three RBFs, while for the construction of the hierarchical sigmoids, only two sigmoid functions are needed.

Each sigmoid has a counterpart, which is for an arbitrary sigmoid Ψ defined as $1-\Psi$ which ensures the partition of unity. On the first hierarchical level, the input space is governed by either Ψ_1 (blue) or $1-\Psi_1$ (yellow), depending on whether the model is left or right of 0.5. In the domain of $1-\Psi_1$, on a second hierarchical level, the sigmoid Ψ_2 and its counterpart $1-\Psi_2$ are added. To obtain the final validity functions, all hierarchically stacked sigmoids are multiplied by each other. The hierarchy of the given example is shown in Fig. 2.17. The leaves of the tree (indicated by the squares) correspond to the three constructed validity functions using different sigmoid nodes, which are indicated by the circles. This idea of hierarchical sigmoids will be used for the HILOMOT construction algorithm explained in Sect. 3.2.2.

2.2.3 Local Model Networks

Until now, all explained neural network structures take the form

$$\hat{y} = \sum_{j=1}^{n_m} \theta_j \Phi_j \left(\underline{u}, \underline{\theta}_j^{[nl]} \right) , \tag{2.53}$$

where the model output is computed by a weighted sum of activation or validity functions. In the case of NRBFs or hierarchical sigmoids as validity functions, the partition of unity holds. The neural network output can thus be seen as a smooth blending between the linear weights θ_j. Therefore, the output is piecewise constant with the smoothness of the transitions determined by the standard deviation of the RBFs or the κ constant in the sigmoid case. To increase modeling capability, instead of using constant weights θ_j, one can choose local linear models (LLMs) $L_j\left(\underline{u}, \underline{\theta}_j\right)$ to construct a neural network. In this case, piecewise linear models are blended instead of constants. This architecture is called local model network (LMN). The linear parameters θ_j are replaced by LLMs $L_j\left(\underline{u}, \underline{\theta}_j\right)$ so that (2.53) turns into

$$\hat{y} = \sum_{j=1}^{n_m} L_j\left(\underline{u}, \underline{\theta}_j\right) \Phi_j\left(\underline{u}, \underline{\theta}_j^{[nl]}\right) . \tag{2.54}$$

The LLMs depend on a set of linear parameters $\underline{\theta}_j$ and also on the location within the input space $\underline{u}$ as

$$L_j\left(\underline{u}, \underline{\theta}_j\right) = \theta_{0j} + \underline{\theta}_j^T \underline{u} . \tag{2.55}$$

The term *linear* might be misleading, but it will be used in this thesis as it is the established terminology in literature. More accurate, though, would be the word *affine* local model as the local model includes an offset θ_{0j}. An illustrative example of a NRBF network in contrast to an LMN can be found in Appx. A.2.

Advantages of LLMs are:

- Higher flexibility in their respective regions of validity in contrast to a single parameter, leading to a smaller number of operating regimes to obtain a satisfactory model quality.
- They have an intuitive interpretation and are easy to understand. They can be seen as linearizations of the nonlinear system in different operating points [139]. By making a weighted combination of these LLMs, one tries to accurately describe the complete nonlinear behavior [86].
- Linear system theory is well developed.
- They are widely used by engineers [139].

Two-dimensional Illustration of an LMN An LMN in a 2-dimensional input space ($n_p = 2$) with three LLMs ($n_m = 3$) is considered. In Fig. 2.18, the RBFs Ψ_j of the local models (upper left-hand side) and the LLMs L_j (upper right-hand side) are shown. After normalizing the RBFs Ψ_j, one obtains the NRBFs (the validity functions) Φ_j (lower left-hand side). The last step is the multiplication of the validity functions with their corresponding LLMs and then the summation of all weighted

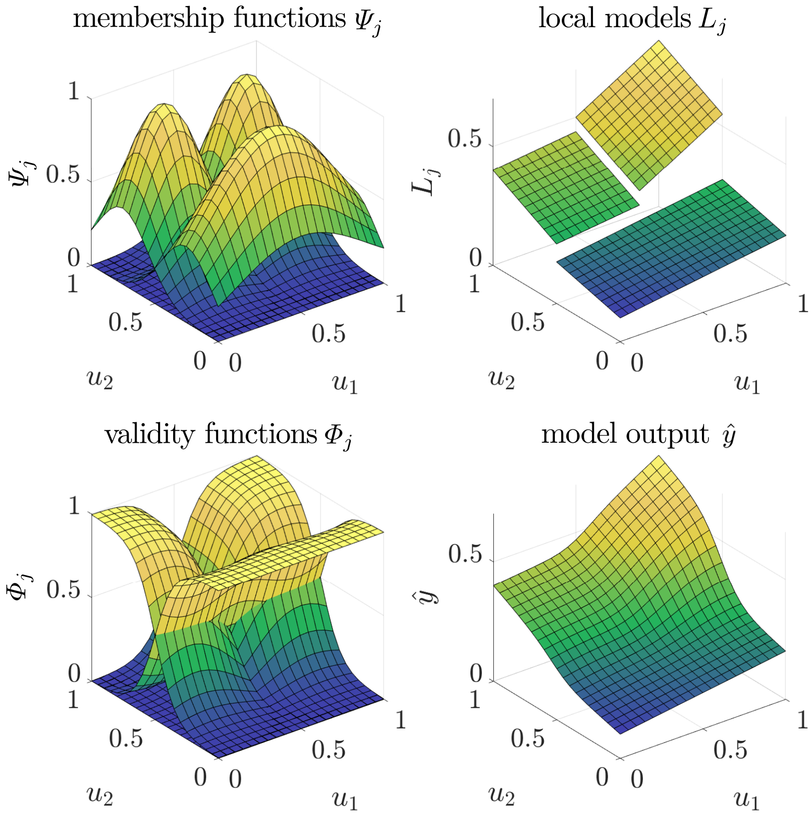


Figure 2.18: Membership functions Ψ_j (upper left), local linear models L_j (upper right), validity functions Φ_j (lower left), and model output $\hat{y}$ (lower right) for an LMN with $n_p = 2$ and $n_m = 3$.

local models (WLMs) to finally obtain the model output $\hat{y}$ (lower right-hand side).

2.3 Nonlinear Optimization

Once a model structure (in our case usually a parametric model) is chosen, the model parameters have to be estimated (also referred to as model *training*). In some nonlinear input-output models, it may be possible to employ linear optimization. For many nonlinear dynamic models, though, the arising optimization problem for parameter estimation is nonlinear in its parameters (as is also true for the LMSSN).

Nonconvex and thus nonlinear optimization problems may have many local optima, so it cannot be ensured that the global optimum is reached after optimization. An analytical solution usually does not exist, making it necessary to employ an iterative algorithm. This also means that an initial guess for the parameter vector is necessary, significantly affecting the local optimum the algorithm converges to [9]. For a few parameters, a direct search algorithm might be used. In those cases, the loss function is simply evaluated at different points of the parameter space, either in a grid-based manner or randomly chosen. A significant disadvantage, though, is the exponentially growing number of evaluation points with an increasing number of parameters [37]. This phenomenon, known as the *curse of dimensionality*, makes it infeasible to apply those methods if the number of parameters is high. Probably the most successful direct *local* method is the downhill simplex method [37]. A simplex is a set of $n_p + 1$ points in a n_p-dimensional input space, which forms for example in $n_p = 2$ dimensions, a triangle. The vertex with the highest loss function value is determined, which is then reflected in the centroid of the other n_p vertices, thus forming a new simplex [37]. In this way, the simplex progresses towards a local optimum. Note that some additional rules need to be established to prevent the algorithm from being trapped. In general, direct methods have slow convergence and are popular mainly because of their relative simplicity to understand and implement [89].

An alternative is gradient descent algorithms. This is the most common and important class of nonlinear local optimization techniques [89] and by far the most popular optimization technique with neural networks [109] and other higher-dimensional problems. Gradient-based algorithms calculate iteratively a new parameter vector $\underline{\theta}(i+1)$ based on the current parameter vector $\underline{\theta}(i)$ corrected in the direction $\underline{p}(i)$ scaled by a step size $\eta(i)$ as

$$\underline{\theta}(i+1) = \underline{\theta}(i) - \eta(i)\underline{p}(i)\,. \tag{2.56}$$

The index i stands for the i-th parameter update, also called the i-th epoch[6]. The direction $\underline{p}(i)$, in which the parameter vector is adjusted, is calculated by

$$\underline{p}(i) = \underline{R}\,\underline{g}(i)\,, \tag{2.57}$$

where $\underline{g}(i)$ is the gradient of the loss function $I(\underline{\theta}(i))$ regarding the parameter vector

$$\underline{g}(i) = \frac{\partial I(\underline{\theta}(i))}{\partial \underline{\theta}(i)} \tag{2.58}$$

[6] Note that the i-th parameter update and i-th epoch only coincide for training in batch mode, which means that all data points are used for a single parameter update. This setup will be used for ease of notation within Sect. 2.3.1-2.3.3. For sample adaption and mini batches (explained in Sect. 2.3.4) a parameter update is performed on a subset of the training data, leading to multiple parameter updates per epoch.

scaled and rotated by some matrix $\underline{R}(i)$. Loosely speaking, one follows the (corrected) slope of the surface created by the loss function downhill until a valley is reached.

The straightforward choice $\underline{R}(i) = \underline{I}$ yields a parameter update in the *steepest descent* direction (Sect. 2.3.1), which is opposite to the gradient $\underline{g}(i)$. There are many further refinements that can be applied to the correction term for more robustness and faster convergence. Among the most popular nonlinear optimization algorithms are *Newton's method*, *Quasi-Newton methods*, and the *adaptive moment estimation (ADAM) method*. Those methods do not require any special structure for the loss function an can thus be categorized as general nonlinear optimization algorithms (for all three see Sect. 2.3.1). In case of a sum of squared errors loss function, the *Gauss-Newton method* and the *Levenberg-Marquardt method* can be employed (see Sect. 2.3.2). For an in-depth study of different nonlinear optimization algorithms, see [112, 59].

2.3.1 General Nonlinear Optimization

We will first consider optimization methods that do not impose any special structure on the loss function.

Steepest Descent As already mentioned, if $\underline{R}(i) = \underline{I}$ is chosen in (2.57) for the parameter update direction, this leads to the steepest descent parameter update equation

$$\underline{\theta}(i+1) = \underline{\theta}(i) - \eta(i)\underline{g}(i)\,. \tag{2.59}$$

The parameter update is exactly opposite to the gradient, yielding for sufficiently small $\eta(i)$ the direction which gives the greatest loss function reduction.

The step size $\eta(i)$ can either be fixed, varying over the number of epochs, or optimized via line search. The first two alternatives have the advantage of being simple and that no additional optimizations have to be done. The line search finds by a univariate parameter optimization the optimal step size at each epoch i in the search direction of $\underline{p}(i)$. Two subsequent problems need to be dealt with for line search: first, finding the interval in which the optimum is located, and second, an interval reduction mechanism. For further detail regarding those problems, refer to [112].

The major advantages of the steepest descent method are listed here [89, 96]:

- easy to understand
- easy to implement
- large region of convergence to a (local) minimum
- no requirement of second-order derivatives
- linear computational complexity
- linear memory requirement complexity.

On the other hand, there are serious drawbacks [50, 89]:

- slow convergence
- affected by a linear transformation of parameters
- generally requires an infinite number of iterations to solve a linear optimization problem.

Improvements to Steepest Descent There are many ways in which the steepest descent method and the direction of the update step can be improved. Second-order methods (like Newton's method) incorporate information about the curvature of the loss function besides the gradient into the parameter updates [3]. This requires the computation of second-order derivatives (the Hessian matrix) and its inversion, which is computationally very expensive. Quasi-Newton methods have been developed to approximate the Hessian (or its inverse) with gradient information and alleviate the computational demand while still providing fast convergence [112].

Another way to improve performance is by introducing so-called momentum terms. The idea is to memorize past gradients and take a parameter update step as a weighted combination of current and past gradients. As an analogy, one can think of a ball rolling downhill the loss function surface. Since it gathered momentum downhill, it will not stop right away at a pit (local optimum) but will use its momentum to possibly overcome a neighboring small hill in the surface to reach an even deeper pit (a better local optimum). Among the most popular optimization algorithms that use the idea of momentum is the ADAM optimizer [59].

Newton's method, the Quasi-Newton method, and ADAM are explained in more detail in Appx. A.3.

2.3.2 Nonlinear Least Squares Optimization

Some optimization methods make use of the structure of the loss function. The by far most commonly used loss function is the sum of squared errors

$$I(\underline{\theta}) = \sum_{k=1}^{N} e^2(k, \underline{\theta})\,, \tag{2.60}$$

where $e(k, \underline{\theta})$ denotes the model error per data sample (or in the case of dynamical modeling the time step) k, which is calculated by $e(k, \underline{\theta}) = y(k) - \hat{y}(k, \underline{\theta})$. The choice of the squared errors loss function is optimal (in the maximum likelihood sense) if the noise is Gaussian distributed [89].

If the model $\hat{y}(k, \underline{\theta})$ is linear in the parameters, (2.60) can be solved by linear least squares. If $\hat{y}(k, \underline{\theta})$ has a more complicated (nonlinear) dependency on the parameters, (2.60) is known as the nonlinear least squares problem. In the following, two algorithms will be explained, which use such a nonlinear least squares problem formulation. As the Levenberg-Marquardt algorithm will be commonly employed for the optimization of the LMSSN, the nonlinear least squares methods will be explained in some detail.

Before we start, some notation for the subsequent sections is introduced. Equation (2.60) can be written in a matrix-vector form as

$$I(\underline{\theta}) = \underline{e}^T \underline{e} \qquad \text{with} \qquad \underline{e} = \left[e(1, \underline{\theta}), e(2, \underline{\theta}), \ldots, e(N, \underline{\theta})\right]^T\,. \tag{2.61}$$

The gradient g_j of the loss function with respect to parameter θ_j is calculated by

$$g_j = \frac{\partial I(\underline{\theta})}{\partial \theta_j} = 2 \sum_{k=1}^{N} e(k) \frac{\partial e(k)}{\partial \theta_j} = 2 \underline{e}^T \frac{\partial \underline{e}}{\partial \theta_j}\,. \tag{2.62}$$

With the definition of the Jacobian

$$\underline{J} = \begin{bmatrix} \frac{\partial e(1,\underline{\theta})}{\partial \theta_1} & \cdots & \frac{\partial e(1,\underline{\theta})}{\partial \theta_{n_\theta}} \\ \vdots & \ddots & \vdots \\ \frac{\partial e(N,\underline{\theta})}{\partial \theta_1} & \cdots & \frac{\partial e(N,\underline{\theta})}{\partial \theta_{n_\theta}} \end{bmatrix}, \tag{2.63}$$

the gradient with respect to the whole parameter vector $\underline{\theta}$ is calculated as

$$\underline{g} = 2\underline{J}^T \underline{e}\,. \tag{2.64}$$

The Hessian $\underline{H}$ contains all second-order derivatives of the loss function with respect to the parameter vector $\underline{\theta}$. One element of the Hessian H_{jl} is calculated as

$$H_{jl} = \frac{\partial^2 I(\underline{\theta})}{\partial\theta_j \partial\theta_l} = 2\sum_{k=1}^{N}\left(\frac{\partial e(k)}{\partial\theta_j}\frac{\partial e(k)}{\partial\theta_l} + e(k)\frac{\partial^2 e(k)}{\partial\theta_j\partial\theta_l}\right)\,, \tag{2.65}$$

and the full Hessian as

$$\underline{H} = 2\underline{J}^T\underline{J} + \underbrace{2\sum_{k=1}^{N} e(k)\frac{\partial^2 e(k)}{\partial\theta_j\partial\theta_l}}_{:=\underline{S}} = 2\underline{J}^T\underline{J} + 2\underline{S}\,. \tag{2.66}$$

Note that one part of the Hessian can be computed purely by information from the Jacobian, while the second part (defined as $\underline{S}$) consists of second-order information.

Gauss-Newton Algorithm If errors are small, the part $\underline{J}^T\underline{J}$ of (2.66) is dominant and $\underline{S} \approx 0$. Therefore, a reasonable approximation of the Hessian can be made by neglecting $\underline{S}$ and approximating the Hessian by

$$\underline{H} \approx 2\underline{J}^T\underline{J}\,. \tag{2.67}$$

Using a quadratic approximation of the loss function leads to the Gauss-Newton update equation

$$\underline{\theta}(i+1) = \underline{\theta}(i) - \eta(i)(\underline{J}^T(i)\underline{J}(i))^{-1}\underline{J}^T(i)\underline{e}(i)\,. \tag{2.68}$$

The Gauss-Newton algorithm can be understood as the nonlinear least squares version of Newton's method. By exploiting the nonlinear least squares loss function structure, it is possible to compute an approximate Hessian only by information of first-order derivatives.

Note that the approximation of the Hessian is only reasonable if $\underline{S} \to 0$. This is true if errors are small[7], which is the case around a (local) optimum. This contrarily means also that the approximation might not work well at the initial parameter

[7] giving this class of algorithms the name *small residual algorithms*

vector, where an optimum is far away. The second term of $\underline{S}$ (the second-order derivatives) can also steer $\underline{S} \rightarrow 0$ if the loss function is only "slightly" nonlinear.

In practice, (2.68) is not solved directly in this formulation, but the linear system of equations is solved

$$\left(\underline{J}^T(i)\underline{J}(i)\right)\underline{p}(i) = \underline{J}^T(i)\underline{e}(i)\,, \tag{2.69}$$

which is also called the normal equation. It can be solved in a numerically stable fashion via the thin (or economy) singular value decomposition (SVD) [43] of $\underline{J}(i)$

$$\underline{J}(i) = \underline{U}(i)\underline{\Sigma}(i)\underline{V}^T(i)\,, \tag{2.70}$$

with $\underline{U}(i)$ of size $(N \times n_\theta)$, $\underline{\Sigma}(i)$ is a diagonal matrix of size $(n_\theta \times n_\theta)$, and $\underline{V}^T(i)$ of size $(n_\theta \times n_\theta)$. This leads to the update direction

$$\underline{p}(i) = \underline{V}(i)\underline{\Sigma}^{-1}(i)\underline{U}^T(i)\underline{e}(i)\,. \tag{2.71}$$

If $\underline{J}(i)$ has not full rank (occurs for over-parametrized models such as a fully populated state space model), then $\underline{\Sigma}^{-1}$ is singular and a truncated SVD can be used to compute (2.71) [96].

Levenberg-Marquardt Algorithm The Levenberg-Marquardt algorithm [65, 74, 82] is an extension of the Gauss-Newton algorithm. It extends (2.68) to

$$\underline{\theta}(i+1) = \underline{\theta}(i) - \eta(i)(\underline{J}^T(i)\underline{J}(i) + \lambda^2\underline{I})^{-1}\underline{J}^T(i)\underline{e}(i)\,, \tag{2.72}$$

where the term $\lambda^2\underline{I}$ is added to the approximate Hessian $\underline{J}^T(i)\underline{J}(i)$. This term has the same effect as employing regularization, like it can be done to the linear least squares problem with ridge regression [89]. Possible problems due to ill-conditioned $\underline{J}^T(i)\underline{J}(i)$ are hereby circumvented.

The interpretation is as follows. Let us first assume a very small λ. In this case, the update formula (2.72) approaches the Gauss-Newton update (2.68). Contrarily, if λ is chosen to be large, then the Levenberg-Marquardt update equation is steered towards the steepest descent update equation (2.59) since

$$(\lambda^2\underline{I})^{-1}\underline{J}^T(i)\underline{e}(i) \sim \underline{g}(i)\,. \tag{2.73}$$

Since the initial parameter vector is likely to be far from the optimum, the Gauss-

Newton method will yield no reliable parameter update because of the approximation of the Hessian made in (2.67). Far away from the optimum, Gauss-Newton might even diverge (for negative $\underline{J}^T(i)\underline{J}(i)$). Therefore, a larger value for λ should be chosen at the start of optimization. If the parameter update is successful, meaning that $I(\underline{\theta}(i+1)) < I(\underline{\theta})$, λ can be reduced by some factor. If the parameter update is unsuccessful ($I(\underline{\theta}(i+1)) \geq I(\underline{\theta})$), λ is increased by some factor until a search direction is found, which yields a drop in the loss function value.

The actual computation of (2.72) is again done via the SVD like

$$\underline{p}(i) = \underline{V}(i)\underline{\Lambda}(i)\underline{U}^T(i)\underline{e}(i)\,, \tag{2.74}$$

with

$$\underline{\Lambda}(i) = \operatorname{diag}\left(\frac{\sigma_1}{\sigma_1^2+\lambda^2}, \frac{\sigma_2}{\sigma_2^2+\lambda^2}, \ldots, \frac{\sigma_{n_\theta}}{\sigma_{n_\theta}^2+\lambda^2}\right)\,. \tag{2.75}$$

Here, $\sigma_1, \sigma_2, \ldots, \sigma_{n_\theta}$ are the singular values of $\underline{J}(i)$. See Appx. A.4 for a detailed derivation and some implementation details of the Levenberg-Marquardt algorithm.

2.3.3 Backpropagation-Through-Time Algorithm

The nonlinear optimization approaches from the previous sections can be applied to any data. In the case of dynamic data, some peculiarities arise in gradient calculations. In the state space framework, the model output $\hat{y}(k)$ is a function of the output equation parameters $\underline{\theta}^{[o]}$, the current input $u(k)$, and the current state vector $\underline{\hat{x}}(k)$

$$\hat{y}(k) = g(\underline{\theta}^{[o]}, \underline{\hat{x}}(k), u(k))\,. \tag{2.76}$$

The state vector $\underline{\hat{x}}(k)$, in turn, depends on the parameters of the state equations $\underline{\theta}^{[s]}$, previous input $u(k-1)$ and the previous state vector $\underline{\hat{x}}(k-1)$ and is in itself recurrent

$$\underline{\hat{x}}(k) = \underline{h}(\underline{\theta}^{[s]}, \underline{\hat{x}}(k-1), u(k-1))\,. \tag{2.77}$$

This can be seen, as the previous state vector $\underline{\hat{x}}(k-1)$ depends, again, on the time step before this

$$\underline{\hat{x}}(k-1) = \underline{h}(\underline{\theta}^{[s]}, \underline{\hat{x}}(k-2), u(k-2)) \tag{2.78}$$

and so on. The unfolded recurrent state equation is depicted in Fig. 2.19.

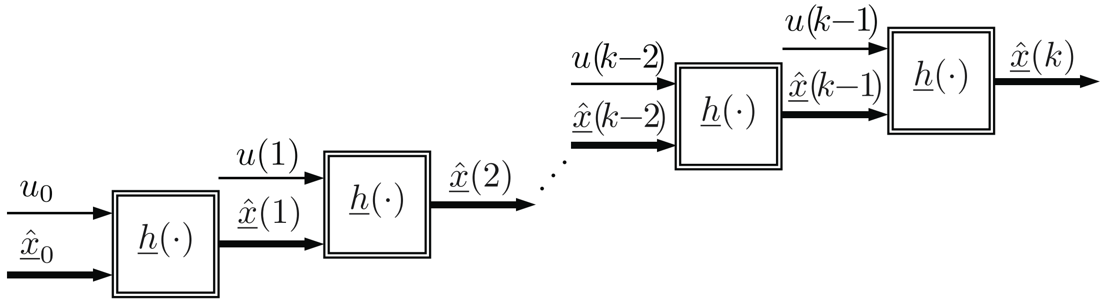


Figure 2.19: Demonstration of the unfolded state equation (see also [89])

Optimizing the state equation parameters $\underline{\theta}^{[s]}$ leads to the *real time-recurrent-learning* algorithm, also known as *simultaneous backpropagation* [89].

This means that the gradient has to be computed by chain rule

$$\frac{\partial \hat{y}(k)}{\partial \underline{\theta}^{[s]}} = \frac{\partial g(\cdot)}{\partial \hat{\underline{x}}(k)} \cdot \frac{\partial \hat{\underline{x}}(k)}{\partial \underline{\theta}^{[s]}} \tag{2.79}$$

with

$$\frac{\partial \hat{\underline{x}}(k)}{\partial \underline{\theta}^{[s]}} = \underbrace{\frac{\partial \underline{h}(\cdot)}{\partial \underline{\theta}^{[s]}}}_{\text{static}} + \underbrace{\frac{\partial \underline{h}(\cdot)}{\partial \hat{\underline{x}}(k-1)} \cdot \frac{\partial \hat{\underline{x}}(k-1)}{\partial \underline{\theta}^{[s]}}}_{\text{dynamic}} \; . \tag{2.80}$$

Equation (2.80) shows that the gradient for the next time step always depends on a static and a dynamic part, which depends on the previous time step. Therefore, this dependence on the previous time steps makes the optimization a pretty complex one.

2.3.4 Gradient Updating Frequency

Until now, we assume that a parameter update is carried out on the whole dataset. This means that the gradient $\underline{g}$ is computed per epoch for the whole dataset $\{u(k), y(k)\}_{k=1}^{N}$ at once. For large N, this can be quite time-consuming. Instead, parameter updates might as well be updated more frequently per epoch, for example, after a specified portion of the dataset “has been seen”, or, in the extreme case, after every data sample.

Batch Gradient Descent For batch gradient descent, the loss function is defined over all data samples as

$$I(\underline{\theta}) = \sum_{k=1}^{N} \left(y(k) - \hat{y}(\underline{\theta}, k)\right)^2 \tag{2.81}$$

and the gradient is calculated as

$$\underline{g}(i) = \frac{\partial I(\underline{\theta}(i))}{\partial \underline{\theta}(i)} \,. \tag{2.82}$$

Within one epoch i of training, there is only a single gradient update. The batch gradient descent is also known by the name *vanilla gradient descent* in the machine learning community.

Stochastic Gradient Descent (or Sample Adaption) For non-dynamical data, a parameter update might as well be carried out for each training sample. So for each epoch of training, there are N gradient calculations and, in turn, N parameter updates. This is called stochastic gradient descent (SGD), as all samples are shuffled at the beginning of each epoch, leading to a highly fluctuating convergence behavior. While SGD is possible for image classification or other static applications, it is not feasible to rearrange the order of individual time steps in dynamic datasets. The solution here is to view one *training sample* not literally as one time step but as a sequence of N_s consecutive time steps. This is what we will call from now on a *sequence* (see Fig. 2.20). Sequences can be shuffled freely during optimization.

The parameter update is thus calculated for the n_s sequences as

$$\underline{\theta}(s+1) = \underline{\theta}(s) - \eta \frac{\partial I(\underline{\theta}(s), \underline{u}^{(s)}, \underline{y}^{(s)})}{\partial \underline{\theta}(s)} \,, \tag{2.83}$$

where

$$\underline{u}^{(s)} = \left[u(k_s), u(k_s+1), \ldots, u(k_s+N_s-1)\right], \tag{2.84}$$

$$\underline{y}^{(s)} = \left[y(k_s), y(k_s+1), \ldots, y(k_s+N_s-1)\right], \tag{2.85}$$

and $k_s = (s-1)N_s + 1$ being the first time step per sequence. One epoch is over

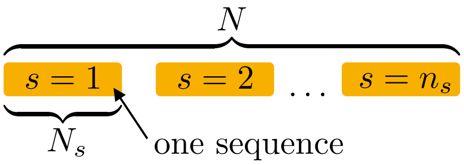


Figure 2.20: Splitting of training data in n_s sequences of length N_s

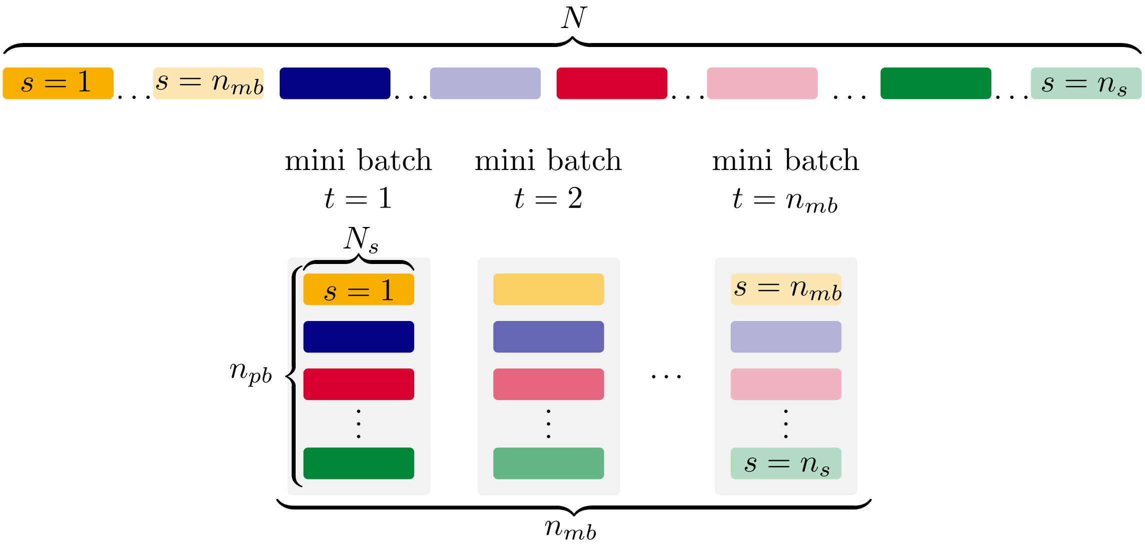


Figure 2.21: Splitting of training data in mini batches. N: total number of time steps in training data, N_s: number of time steps per sequence, n_s: number of sequences, n_{pb}: number of sequences per mini batch, n_{mb}: number of mini batches

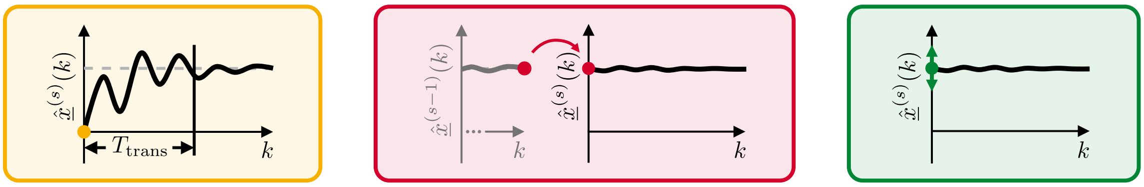
when all $s = 1, \dots, n_s$ sequences have been trained on once[8]. An advantage of SGD is that it might jump to new and potentially better local optima due to the fluctuating gradient updates.

Mini Batch Gradient Descent Now, the best from vanilla gradient descent and SGD is taken to make a compromise between those two, called *mini batch gradient descent*. In contrast to SGD, more data points are used per parameter update. This generally leads to a reduction of variance of the parameter update direction which can lead to more stable convergence. Highly optimized matrix computations may be employed for computational efficiency.

Each mini batch $t = 1, \dots, n_{mb}$ consists of n_{pb} sequences, for which the model output is computed simultaneously (see Fig. 2.21). The loss function is therefore not calculated as in (2.81), but per mini batch t by

$$I^{(t)}(\underline{\theta}) = \sum_{s \in M_t} \sum_{k=1}^{N_s} (y^{(s)}(k) - \hat{y}^{(s)}(k))^2 \,, \tag{2.86}$$

[8] Some notational remarks: note that the gradient calculations per sequence s are subroutines to the gradient calculation per epoch i. The initial parameter vector $\underline{\theta}(s=1) = \underline{\theta}(i)$ and the last parameter update (with the last sequence as input) yields $\underline{\theta}(i+1) = \underline{\theta}(n_s+1)$.


$\hat{\underline{x}}^{(s)}(k)$
T_{trans}
k
$\hat{\underline{x}}^{(s-1)}(k)$
k
$\hat{\underline{x}}^{(s)}(k)$
k
$\hat{\underline{x}}^{(s)}(k)$
k

a) fixed — $\hat{\underline{x}}_0^{(s)} = \underline{0}$

b) handover — $\hat{\underline{x}}_0^{(s)} = \hat{\underline{x}}^{(s-1)}(k_{s-1} + N_{s-1} - 1)$

c) optimize — optimize $\hat{\underline{x}}_0^{(s)}$

Figure 2.22: Different options for initial state vector $\hat{\underline{x}}_0^{(s)}$ of each sequence s (illustrated for a scalar state). The initial state can either be a) chosen as fixed ($= \underline{0}$), b) handed over from the last state of the preceding sequence, or c) optimized. The state of the process is shown as dashed gray line.

where M_t is a set of sequence indices for the t-th mini batch

$$M_t = \{t, n_{mb} + t, 2n_{mb} + t, \ldots, (n_{pb} - 1)n_{mb} + t\} \ . \tag{2.87}$$

The sequences are obtained by partitioning the training data in sequences of length N_s. The superscript $(\cdot)^{(s)}$ indicates the process and model output in sequence s. The mini batch sample size (samples summed over all sequences) is thus calculated by $N_{mb} = N_s n_{pb}$. Common mini batch sizes n_{pb} range from 50 to 256 [109].

Initialization of States When the training data is split into multiple sequences, different options for the initialization of each initial state vector $\hat{\underline{x}}_0^{(s)}$ arise (see Fig. 2.22). The simplest choice is to a) fix it to a constant vector, usually $\underline{0}$. Sequences can be shuffled freely and no care concerning the sequence order has to be taken during optimization. This simple approach has the consequence that each sequence has its own transient phase T_{trans}. This phase can be discarded for optimization (so that the model does not fit the transient effects due to possibly unfavorable initialization), but this means at the same time that valuable data points are lost. Alternatively, the transient is assumed to be short and it is simply accepted that gradient calculations are off during this initial phase.

The second choice is b) a handover of the final state vector of the preceding sequence $\hat{\underline{x}}^{(s-1)}(k_{s-1} + N_{s-1} - 1)$ as the initial state of the current sequence $\hat{\underline{x}}_0^{(s)}$ which is sometimes referred to as cross-batch statefulness. This approach has the advantage that initial data samples do not have to be left out from gradient calculations.[9] On

[9] Only the sequences of the first mini batch cannot be treated in this way. Here, either option a) or c) might be chosen.

the other hand, sequences can no longer be shuffled freely, restricting the stochasticity of the optimization.

The third choice is to c) optimize the initial state vector as free parameters. The advantage of this method is like for the handover that no data points are lost. However, this comes at the cost that additional parameters need to be optimized.

Assessment of Different Gradient Update Strategies How much and in which constellation the data is used per gradient update leads to different properties during model training. Table 2.1 compares different gradient update strategies regarding stochasticity, computational demand, the dimensionality of the optimization problem, and error-prone gradients due to transient effects. Since handover is never possible for the first mini batch, it is listed in combination with either fixed or optimized initial states for the aforementioned first mini batch. Training with all data in one batch is fully deterministic and computationally very demanding since no parallelization is done. If the initial state is fixed, the transient has to fade only once, usually

Table 2.1: Assessment of different gradient update strategies. ++: very high, +: high, o: neutral, −: low, −−: very low. Single sequence mini batch and multiple sequence mini batch, see Fig. 2.20 and Fig. 2.21, respectively. For initialization choices, see Fig. 2.22. If two initialization options are listed, the first is for the first mini batch and the second is for all other mini batches.

	Batch		Single sequence mini batch			Multiple sequence mini batch		
Initial state ...	fixed	optimize	fixed	fixed/handover	optimize/handover	fixed	fixed/handover	optimize/handover
Stochasticity	−−	−−	++	o	o	+	−	−
Computational demand for loss function evaluation	++	++	++	++	++	−−	−−	−−
Additional dimensions of optimization problem	−−	+	−−	−−	+	−−	−−	++
Error-prone gradients due to transient effects	−	−−	++	−	−−	+	−	−−

leading to robust gradient estimates. The initial states can be optimized, increasing the dimensionality of the optimization problem. Single sequence mini batches with fixed initial states grant a lot of stochasticity while having the same computational demand as batch training. Since every sequence does now have a transient phase, it is likely to possess error-prone gradients. This can be alleviated by introducing the state handover, which reduces this error while also decreasing stochasticity. The computational demand can be reduced most effectively through multiple sequences per mini batch. When the state is transferred between mini batches, the gradients are less error-prone. This goes again hand in hand with reduced stochasticity. If the initial states of all sequences in the first mini batch are optimized, the dimensionality of the optimization problem increases, making optimization computationally more demanding.

2.3.5 Reflection on Nonlinear Optimization

The choices for nonlinear optimization methods are plentiful. Depending on the problem type, available data, and the number of parameters, different strategies are more favorable than others. A clear statement which algorithm is best in which case is not possible. Discovering the optimal choices that need to be made is tedious and time-consuming and more often reflect personal preferences than objective arguments. It is even argued that local nonlinear optimization methods are more art than technology [9].

2.4 Model Complexity

How can it be assured that the estimated model does not only describe the training data well but also *generalizes* well (that is, it performs well on unseen test data)? One can tackle this problem in two ways. Either, on the level of model structure. Then the process of selecting a model that performs best on test data is called *complexity selection* [83]. Or, one can tackle the problem on the level of model parameters. Reducing the complexity of a model by influencing the optimizable parameters is called *regularization*. The notion of sensible model complexity is motivated by the bias/variance tradeoff, which is explained in the following.

2.4.1 Bias/Variance Tradeoff

Let us consider a process with additive noise at the process output as

$$y(u) = y_u(u) + n\,, \tag{2.88}$$

where $y_u(u)$ is the undisturbed process and n denotes i.i.d. noise with mean zero and variance σ^2. A model $\hat{y}(u)$ is trained to fit the process. For assessment of model quality, the expectation of the squared error may be considered in a probabilistic framework, which yields

$$\mathbb{E}(e^2) = \mathbb{E}((y - \hat{y})^2) = \mathbb{E}((y_u - \hat{y})^2) + \sigma^2\,. \tag{2.89}$$

Note that the cross-terms $\mathbb{E}((y_u - \hat{y})n)$ vanish because the noise is uncorrelated with the process and model outputs. The dependency on the process input is left out for readability. The first term describes the difference between true (but not measurable) process output y_u and model output $\hat{y}$, while the second term is the noise variance. The first term shall be further analyzed because it depends on the model, while the second term, the noise variance, is inherent to the data and therefore not reducible.

Decomposing the first term yields [89, 68]

$$\mathbb{E}((y_u - \hat{y})^2) = \underbrace{(y_u - \mathbb{E}(\hat{y}))^2}_{\text{bias}^2} + \underbrace{\mathbb{E}((\hat{y} - \mathbb{E}(\hat{y}))^2)}_{\text{variance}}\,. \tag{2.90}$$

One can see that the model error consists of two parts: a bias term and a variance term. The bias term describes the discrepancy between the true (but unknown) process and the best model available in the (possibly too simple) model class. The variance error measures the distance between actually estimated model and the best available model of the model class. It is a measure for the uncertainty associated with parameter estimation [70]. Figure 2.23 illustrates the tradeoff that needs to be made between bias and variance. For models with low complexity, the variance error is low, while the bias error is high. With increasing complexity, the bias decreases and the variance increases. The optimal model is found when the sum of bias and variance error is minimal.

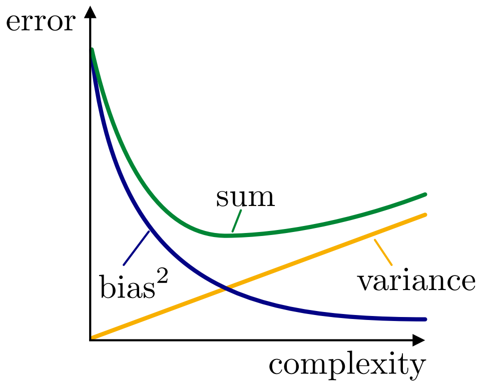


Figure 2.23: Schematic illustration of the bias/variance tradeoff. The bias error decreases with increasing model complexity, but the variance error increases.

2.4.2 Model Complexity Selection

For model complexity selection, three common strategies are introduced.

Train/Test Split A quite simple approach to ensure generalization and find a good bias/variance tradeoff is to divide the dataset into training, validation, and test datasets. The model parameters are fit on training data, model selection is performed on validation data, while the final model quality is assessed on the held back test dataset. This method is appealing due to its simplicity but comes at the cost that not all data points can be used for model training.

Cross-validation An alternative is to split the dataset into S parts (also called folds) and use $S-1$ parts for training and one part for validation. This procedure can be carried out with S different combinations of data folds. In this way, all available data is employed and by adding all validation errors, a reliable estimate for the performance on test data is found [89]. This method is called *S-fold cross-validation*. If the number of folds is chosen equal to the number of samples N, then the procedure is called *leave-one-out cross-validation*. Of course, now the training procedure has to be carried out N times.

Cross-validation techniques are feasible when model training is computationally not too demanding. As parameters are usually optimized nonlinearly in this work, which is computationally very demanding, cross-validation techniques are not employed.

Information Criteria A third alternative for complexity selection is by means of information criteria. Here, a complexity penalty on the training loss is introduced, which gets larger with an increasing number of parameters. These information criteria are derived from different statistical assumptions. Among the most popular are the corrected Akaike information criterion ($\mathrm{AIC_c}$) and corrected Bayesian information criterion ($\mathrm{BIC_c}$) [15].

2.4.3 Regularization

Regularization techniques reduce model complexity without changing the nominal number of parameters. Therefore, the model might be less flexible than it appears by considering the number of parameters alone [89]. Two common regularization techniques are explained in the following.

Penalty Terms One way model parameters are regularized is through additional penalty terms in the loss function. Those terms might penalize the non-smoothness of the learned function. A common choice here is ridge regression [50], which penalizes the magnitude of each individual parameter by extending the loss function to

$$I(\underline{\theta}, \lambda) = \sum_{k=1}^{N} (y(k) - \hat{y}(k, \underline{\theta}))^2 + \lambda \sum_{i=1}^{n_\theta} \theta_i^2 \,. \tag{2.91}$$

The penalty term pushes parameters towards zero, reducing the effective number of model parameters. The strength of the penalization is controlled by the hyperparameter λ. In terms of Fig. 2.23, model complexity is reduced, decreasing the variance error but increasing the bias error.

Early Stopping Early stopping can be applied in any iterative optimization scheme. The parameters are not trained until convergence. Instead, the error on validation data is monitored during training, and training is stopped when the validation error reaches its minimum. At this point, the best bias/variance tradeoff is found. Typical convergence curves are shown in Fig. 2.24. When the validation error is still decreasing, the model is underfitting. When the validation error rises again, the model is overfitting the training data. The intuition behind this is that the most important parameters (with high sensitivity of the loss) converge faster than unimportant parameters. Therefore, at the best bias/variance tradeoff point, important parameters

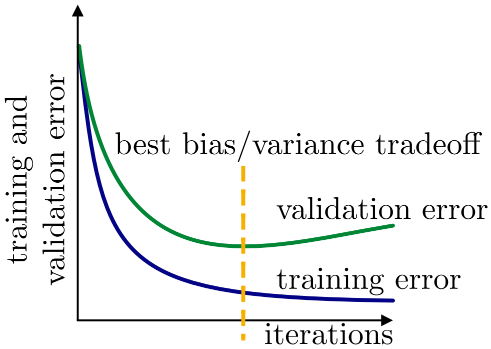


Figure 2.24: Early stopping monitors the validation error during training. When the validation error does not decrease further, training is stopped.

have converged or are close to convergence, while unimportant parameters remain close to their initial value [89].

3 Local Model State Space Network

Parts of the chapter have been published in [124, 125, 126].

In this chapter, the proposed local model state space network (LMSSN) is explained in detail. Let us consider a single-input single-output (SISO)[1] deterministic nonlinear time-discrete state space model

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \underline{h}(\underline{\hat{x}}(k), u(k)) \\ \hat{y}(k) &= g(\underline{\hat{x}}(k), u(k))\,, \end{aligned} \tag{3.1}$$

with the state vector $\underline{\hat{x}}(k) \in \mathbb{R}^{n_x}$, the input $u(k) \in \mathbb{R}$, the model output $\hat{y}(k) \in \mathbb{R}$, the state equation $\underline{h}(\cdot) : \mathbb{R}^{n_x+1} \rightarrow \mathbb{R}^{n_x}$, and the output equation $g(\cdot) : \mathbb{R}^{n_x+1} \rightarrow \mathbb{R}$ at the discrete time step k (see Fig. 3.1). There are, in general, many different approaches for the parametrization and estimation of $\underline{h}(\cdot)$ and $g(\cdot)$. As local model networks (LMNs) have been proven to be an effective architecture for nonlinear function approximation, it seems reasonable to expect high performance also in the state space context. Therefore, we will consider LMNs for state and output equation in different configurations and study this novel approach's characteristics and implications on

[1] Note that we will consider SISO systems for ease of understanding and notation. The extension of the LMSSN to the multiple-input multiple-output (MIMO) case is straightforward. Derivations of transformations and gradient calculations are shown in full generality for the MIMO case in the appendices.

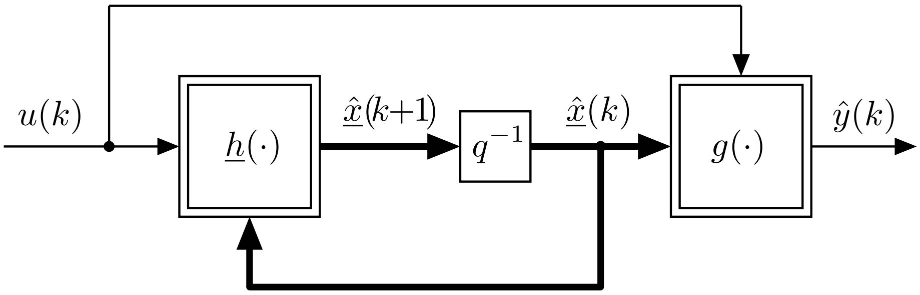


Figure 3.1: SISO nonlinear state space model. Normal weight arrows indicate scalar signal flows, while boldface arrows indicate vector signal flows.

model training. The arising model structure, as well as the associated identification algorithm, is called LMSSN.

The LMSSN is attractive for the following reasons:

- As a starting point for optimization, a linear state space model is employed. This ensures that the worst nonlinear model is at least as good as the best linear model (see Sect. 2.1.2.2).
- The LMSSN makes use of the advantages that come from the state space dynamics realization in contrast to input-output dynamics realizations (see Sect. 2.1.3).
- Imposing certain restrictions, the LMSSN resembles other model structures, like a piecewise affine (PWA) state space model or block-oriented structures like the Wiener or Hammerstein model (see Sect. 3.5).
- The order of the model is the only hyperparameter that *has to* be specified by the user in a black-box setting. Further refinements *can* be made but are not necessary.
- Each state variable might exhibit different dynamic behavior. The LMSSN can model a unique state transition map for each state variable instead of oversimplifying to a linear model or choosing the complexity too high by describing all state variables nonlinearly (example in Sect. 5.3).
- Prior knowledge can be incorporated in the LMSSN structure (example in Sect. 5.3).

The structure of this chapter can be seen in Fig. 3.2. First, the model structure and construction algorithms will be explained, followed by some pivotal steps during the identification of an LMSSN model. Next, some characteristics and relations to other model structures will be investigated. Numerical and computational aspects will be examined at the end of the chapter.

3.1 Model Structure

The LMSSN model can be viewed from two different perspectives. On the one hand, a perspective which we will call the *neural network perspective*. This perspective interprets all mappings as a weighted superposition of local affine models. On the other hand, one can interpret the LMSSN as a time-variant affine state space model

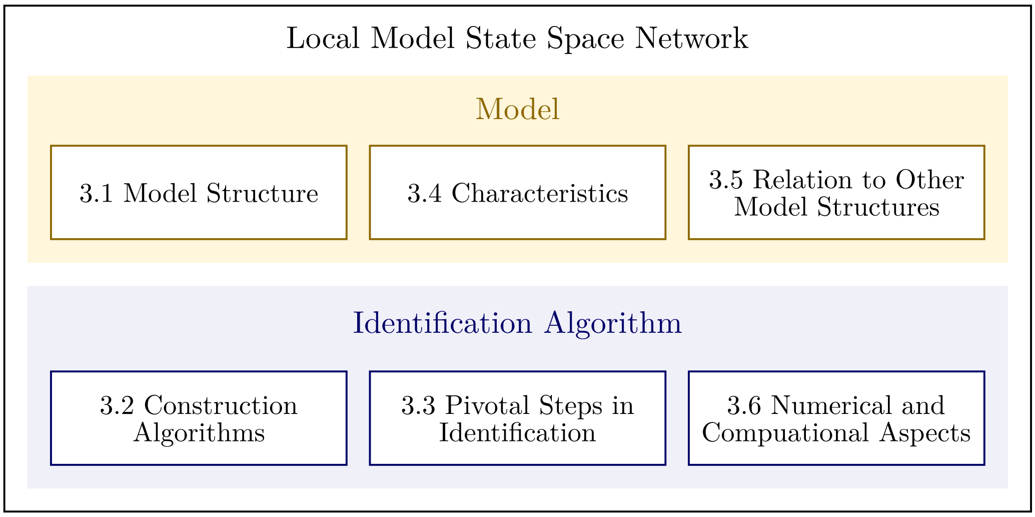

Local Model State Space Network
Model
3.1 Model Structure
3.4 Characteristics
3.5 Relation to Other Model Structures
Identification Algorithm
3.2 Construction Algorithms
3.3 Pivotal Steps in Identification
3.6 Numerical and Compuational Aspects

Figure 3.2: Structure of this chapter

with a distinct scheduling vector $\Phi(k)$. We call this perspective the *system identification perspective*. Depending on the analysis, the former or the latter perspective will be taken.

3.1.1 Neural Network Perspective

The output $\hat{y}(k)$ of an LMN (detached from the context of state space models) is calculated by

$$\hat{y}(k) = \sum_{j=1}^{n_m} \underbrace{L_j\left(\underline{\tilde{u}}(k), \underline{\theta}_j\right)}_{\text{Local Model}} \overbrace{\Phi_j\left(\underline{\tilde{u}}(k), \underline{\theta}_j^{[nl]}\right)}^{\text{Validity Function}} . \tag{3.2}$$

The difference to the previously introduced static LMN in Sect. 2.2.3 is that now the LMN is applied to a dynamic setting, hence the dependence on the time step k. Here, n_m is the number of local models (LMs). The LM L_j depends on an extended input vector $\underline{\tilde{u}}(k)$ and the parameters of the LM $\underline{\theta}_j$. The extended or inner input vector $\underline{\tilde{u}}(k)$ is the input vector to the LMNs. In the context of state space models, this will always be $\underline{\tilde{u}}(k) = [\underline{\hat{x}}^T(k)\ u(k)]^T$. The extended or inner input vector is introduced to distinguish it from the dynamic model input and process input $u(k)$. In principle, any local model structure can be chosen for L_j, but only local linear models (LLMs) are considered, for the given reasons from Sect. 2.2.3. The validity function Φ_j depends on the extended input vector $\underline{\tilde{u}}(k)$ and a set of nonlinear parameters $\underline{\theta}_j^{[nl]}$ (see Sect. 2.2.2). To shorten the notation in all further elaborations, we write $L_j(k)$ and $\Phi_j(k)$ instead of $L_j(\underline{\tilde{u}}(k), \underline{\theta}_j)$ and $\Phi_j(\underline{\tilde{u}}(k), \underline{\theta}_j^{[nl]})$, respectively.

For the incorporation of LMNs in the state and output equation of the nonlinear state space model, a multitude of options arises, which make use of different combinations of (i) banks of multiple-input single-output (MISO) LMNs, (ii) MIMO LMNs, and (iii) affine functions. All three options can, in principle, be employed in state and output equations of a MIMO LMSSN. Since affine functions are not sensible in the state equation (the model would have affine instead of nonlinear dynamic behavior) and a MIMO LMN is not sensible in the output equation (we will only consider the single-output case), four main LMSSN setups arise which are shown in Fig. 3.3 (a)-(d). Those four setups give a general idea and further extensions are easier to grasp. In the end, options, where only part of the state vector is combined in one LMN and other state variables are set as separate LMNs, are possible. We will start with the most general structure (a) and, by restricting certain properties of the LMNs, obtain the models (b)-(d).

State Equation In the state equation $\underline{h}(\cdot)$, the updated state vector can be calculated by a bank of MISO LMNs (see Fig. 3.4). In this case, there exist n_x LMNs, therefore one LMN per state variable (one entry of the state vector). The i-th state variable is calculated by

$$\hat{x}_i(k+1) = \sum_{j=1}^{n_{m_i}} L_{i,j}^{[s]}(k) \Phi_{i,j}^{[s]}(k)\,, \tag{3.3}$$

where the LMN is made up of n_{m_i} LMs. The number of LMs for the state equation sums up to $n_n = \sum_{i=1}^{n_x} n_{m_i}$. The number of parameters in the state equation adds up to $n_\theta = (n_x + 2) \cdot n_n$. This approach makes it possible to define individually how many LMs are needed for satisfactory model quality for each state variable. Those equation structures will be used in models (a) and (b) of Fig. 3.3.

Another option is to model the state equation $\underline{h}(\cdot)$ with a single MIMO LMN as

$$\hat{x}_i(k+1) = \sum_{j=1}^{n_m} L_{i,j}^{[s]}(k) \Phi_j^{[s]}(k). \tag{3.4}$$

The difference between (3.4) and (3.3) is that only one MIMO LMN is needed instead of a bank of MISO LMNs. This means that $n_{m_i} = n_m$ and $\Phi_{i,j}^{[s]}(k) = \Phi_j^{[s]}(k)$ for all state variables $i = 1, \ldots, n_x$. The number of LMs equals $n_n = n_m$ and the number of parameters sums up to $n_\theta = (n_x + 2) \cdot n_x \cdot n_m$. Those equation structures will be used in models (c) and (d) of Fig. 3.3.

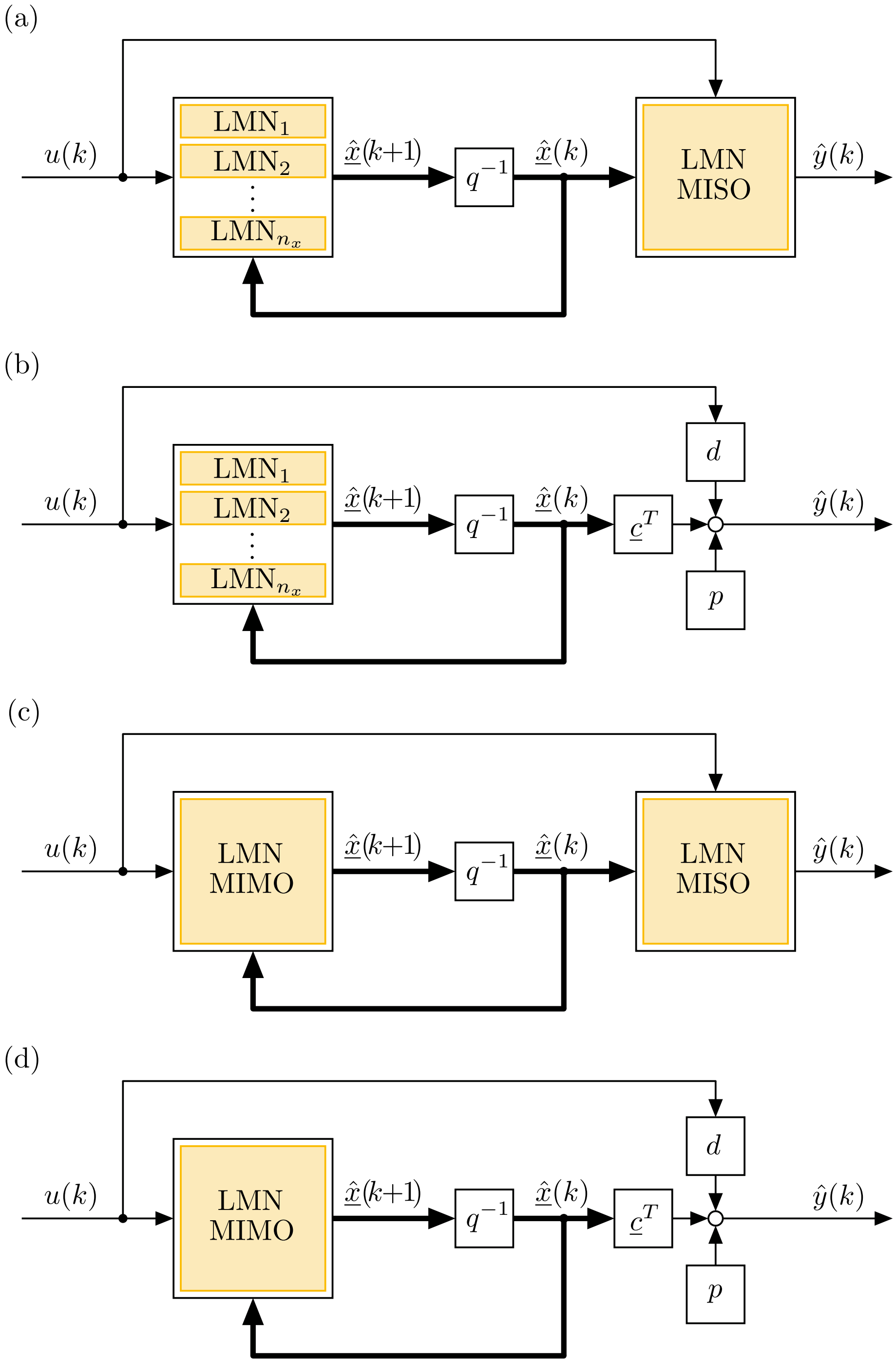


Figure 3.3: LMSSN structures. Different combinations of LMNs in state and output equation.
(a) Bank of MISO LMNs in state equation, MISO LMN in output eq.
(b) Bank of MISO LMNs in state equation, affine output equation
(c) MIMO LMN in state equation, MISO LMN in output equation
(d) MIMO LMN in state equation, affine output equation

Output Equation The output equation can be made up of one MISO LMN, leading to the output equation

$$\hat{y}(k) = \sum_{m=1}^{n_o} L_m^{[o]}(k) \Phi_m^{[o]}(k)\,, \tag{3.5}$$

used in models (a) and (c) of Fig. 3.3. Or, by choosing $n_o = 1$, which means that $\Phi_1^{[o]}(k) = 1$ for all $k = 1, \ldots, N$, one obtains the affine output equation

$$\hat{y}(k) = L^{[o]}(k)\,, \tag{3.6}$$

used in models (b) and (d) of Fig. 3.3.

As already mentioned, it is also possible to separate the state vector into two parts where one part is a set of state variables represented by a MIMO LMN and another part is set by individual MISO LMNs. This choice grants the possibility to adjust the model flexibility precisely to the user's needs.

Decomposing the Network Structure We want to decompose the most general type of LMSSN as is depicted in Fig. 3.3 (a). The state vector $\underline{\hat{x}}(k)$ and the dynamic input $u(k)$ are together the *inner input vector* $\underline{\tilde{u}}(k) = [\underline{\hat{x}}(k)^T \; u(k)]^T$. This inner input vector can be thought of as *operating point variable* [68] or *scheduling variable* [139]. It is the input to a set of n_x LMNs (see Fig. 3.4). The updated state vector $\underline{\hat{x}}(k+1)$, which consists of the state variables $\underline{\hat{x}}(k+1) = [\hat{x}_1(k+1),\, \hat{x}_2(k+1),\, \ldots,\, \hat{x}_{n_x}(k+1)]^T$, is delayed by the time-shift operator q^{-1} one time step to obtain $\underline{\hat{x}}(k)$. After that, the state vector is, on the one hand, internally fed back to the input and, on the other hand, used in the output equation. In the latter case, this means that $\underline{\hat{x}}(k)$ is multiplied by the parameters $\underline{c}^T$ and added to the input weighted with d and added to the offset p to finally obtain the output $\hat{y}(k)$.[2]

Next, the inside of LMN$_i^{[s]}$ (yellow box in Fig. 3.4) is examined. The structure of LMN$_i^{[s]}$ can be seen in Fig. 3.5. Here, the inner input $\underline{\tilde{u}}(k)$ enters n_{m_i} different weighted local models (WLMs). Each[3] WLM$_{i,j}^{[s]}$ returns a LLM $L_{i,j}^{[s]}$ multiplied by a validity function $\Phi_{i,j}^{[s]}$, which are then summed up to obtain the state variable $\hat{x}_i(k+1)$.

The inside of a WLM$_{i,j}^{[s]}$ (blue box) of Fig. 3.5 is shown in Fig. 3.6. Here, the inner input $\underline{\tilde{u}}(k)$ enters a validity function and a respective LLM (in this case, LLM$_{i,j}^{[s]}$

[2] Note that thin arrows indicate scalar quantities and bold arrows vectorial quantities.

[3] Indices i, j are used if the j-th WLM is considered within the i-th LMN.

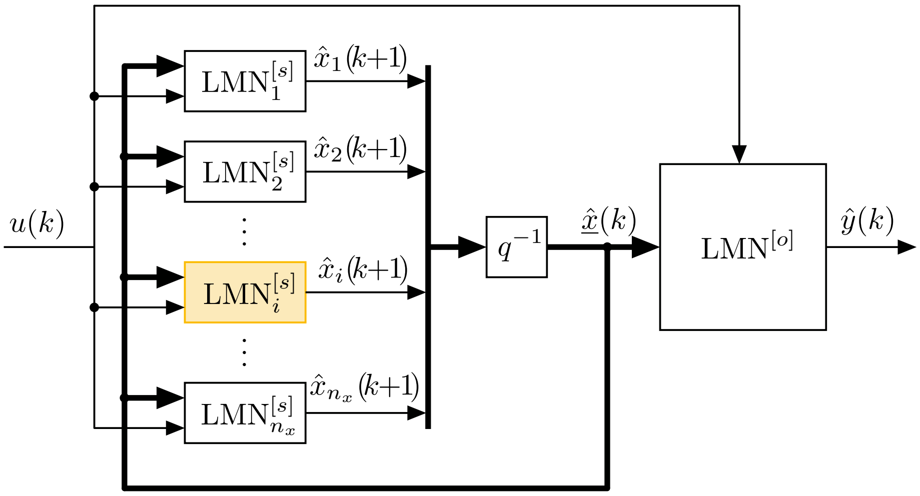


Figure 3.4: LMSSN with MISO state and output equation. Each state variable is modeled by a separate LMN.

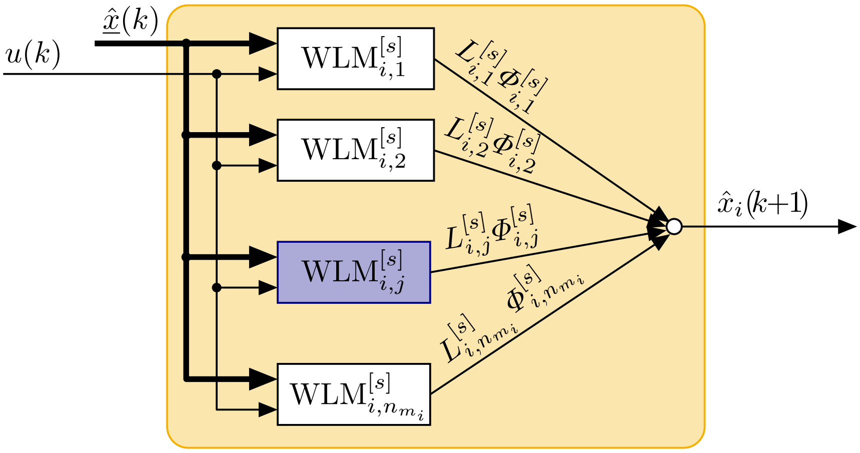

$\underline{\hat{x}}(k)$
$u(k)$
$\mathrm{WLM}_{i,1}^{[s]}$
$\mathrm{WLM}_{i,2}^{[s]}$
$\mathrm{WLM}_{i,j}^{[s]}$
$\mathrm{WLM}_{i,n_{m_i}}^{[s]}$
$L_{i,1}^{[s]}\Phi_{i,1}^{[s]}$
$L_{i,2}^{[s]}\Phi_{i,2}^{[s]}$
$L_{i,j}^{[s]}\Phi_{i,j}^{[s]}$
$L_{i,n_{m_i}}^{[s]}\Phi_{i,n_{m_i}}^{[s]}$
$\hat{x}_i(k+1)$

Figure 3.5: Block diagram of $\mathrm{LMN}_i^{[s]}$

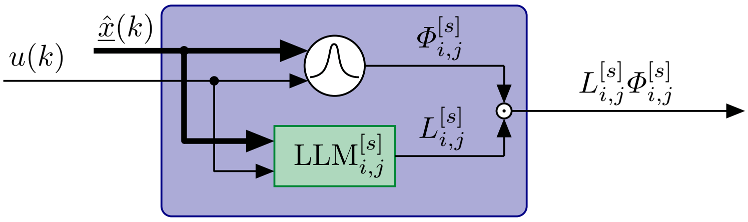


Figure 3.6: Block diagram of $\mathrm{WLM}_{i,j}^{[s]}$

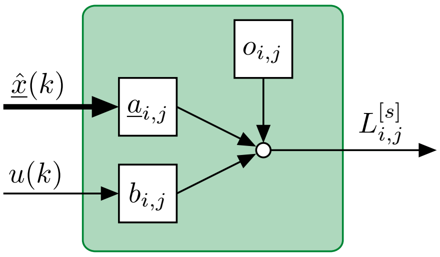


Figure 3.7: Block diagram of $\mathrm{LLM}_{i,j}^{[s]}$

returning $L_{i,j}^{[s]}$). The product $L_{i,j}^{[s]}\Phi_{i,j}^{[s]}$ is the output of the j-th WLM of the i-th LMN.

The inside of a $\mathrm{LLM}_{i,j}^{[s]}$ (green box in Fig. 3.6) is illustrated in Fig. 3.7. The following quantities are added up within the local models: the offset parameter $o_{i,j}$, the input weighted by $b_{i,j}$, and the state vector weighted by $\underline{a}_{i,j}^T$.

The LMSSN is therefore represented by the following state[4] and output equation

$$\hat{x}_i(k+1) = \sum_{j=1}^{n_{m_i}} \left[o_{i,j} + \underline{a}_{i,j}^T \underline{\hat{x}}(k) + b_{i,j} u(k)\right] \Phi_{i,j}^{[s]}(k) \tag{3.7}$$

$$\hat{y}(k) = \sum_{m=1}^{n_o} \left[p_m + \underline{c}_m^T \underline{\hat{x}}(k) + d_m u(k)\right] \Phi_m^{[o]}(k)\,. \tag{3.8}$$

The parameters of the state equation LLMs are gathered in $\underline{\theta}^{[s]}$, which includes all $o_{i,j}$, $b_{i,j}$, and $\underline{a}_{i,j}^T$. The parameters of the output equation LLM will be gathered in $\underline{\theta}^{[o]}$, which includes p_m, d_m, and $\underline{c}_m^T$. All optimizable parameters are stacked into the overall parameter vector $\underline{\theta} = [\underline{\theta}^{[s]^T}, \underline{\theta}^{[o]^T}]^T$.

The state equation of a first-order LMSSN with three LLMs is shown in Fig. 3.8. Note that the colors for LLMs, WLMs, and LMN are chosen the same way as for the block diagrams from Fig. 3.4 - Fig. 3.7.

3.1.2 System Identification Perspective

The other perspective that can be taken on the LMSSN is from the side of system identification. In this view, the LMSSN is perceived as a time-varying affine state

[4] In (3.7) only the i-th state variable of all n_x state variables is shown for better readability.

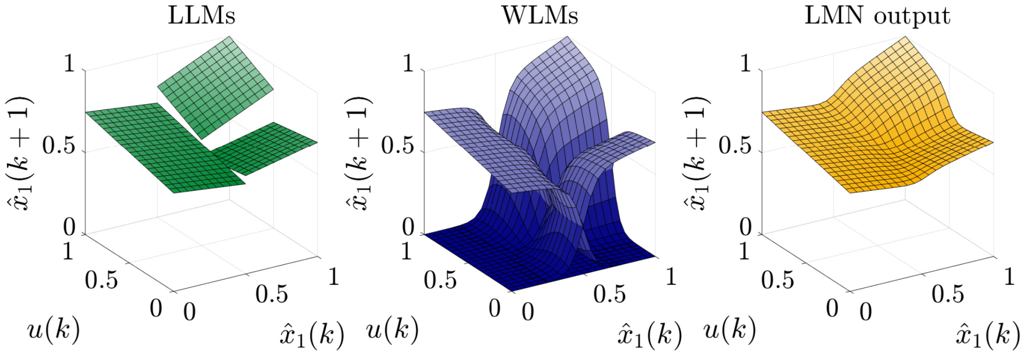


Figure 3.8: Neural network perspective on state equation of a first-order LMSSN with three LLMs. LLMs, WLMs, and LMN are color-coded the same way as the block diagrams from Fig. 3.4 - Fig. 3.7.

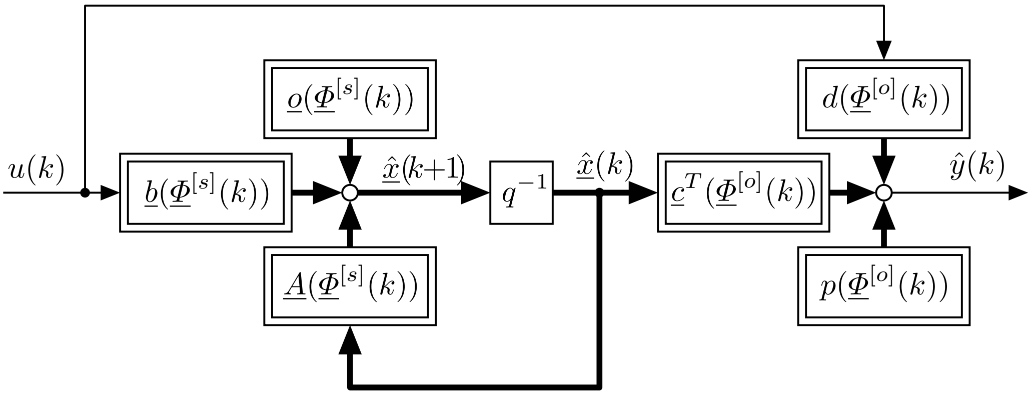

$u(k)$
$\underline{b}(\underline{\Phi}^{[s]}(k))$
$\underline{o}(\underline{\Phi}^{[s]}(k))$
$\underline{A}(\underline{\Phi}^{[s]}(k))$
$\hat{\underline{x}}(k+1)$
q^{-1}
$\hat{\underline{x}}(k)$
$\underline{c}^T(\underline{\Phi}^{[o]}(k))$
$d(\underline{\Phi}^{[o]}(k))$
$p(\underline{\Phi}^{[o]}(k))$
$\hat{y}(k)$

Figure 3.9: System identification perspective on LMSSN. It can be seen as a parameter-varying affine state space model. Each parameter depends on the validity functions as scheduling variables.

space model as

$$\begin{aligned}\hat{\underline{x}}(k+1) &= \underline{o}(\underline{\Phi}^{[s]}(k)) + \underline{A}(\underline{\Phi}^{[s]}(k))\hat{\underline{x}}(k) + \underline{b}(\underline{\Phi}^{[s]}(k))u(k) \\ \hat{y}(k) &= p(\underline{\Phi}^{[o]}(k)) + \underline{c}^T(\underline{\Phi}^{[o]}(k))\hat{\underline{x}}(k) + d(\underline{\Phi}^{[o]}(k))u(k)\,,\end{aligned} \tag{3.9}$$

with the validity functions $\underline{\Phi}^{[s/o]}(k)$ as a scheduling variable (see Fig. 3.9). Only the parameters of the model

$$\underline{\Theta}(\underline{\Phi}^{[s/o]}(k)) = \begin{bmatrix} \underline{o}(\underline{\Phi}^{[s]}(k)) & \underline{A}(\underline{\Phi}^{[s]}(k)) & \underline{b}(\underline{\Phi}^{[s]}(k)) \\ p(\underline{\Phi}^{[o]}(k)) & \underline{c}^T(\underline{\Phi}^{[o]}(k)) & d(\underline{\Phi}^{[o]}(k)) \end{bmatrix} \tag{3.10}$$

vary over time. The way $\underline{\Phi}^{[s/o]}(k)$ is chosen makes the LMSSN a new approach (see Sect. 3.4).

Decomposing the Parameter-varying Structure Let us consider the parameter matrix $\underline{\Theta}(\underline{\mathit{\Phi}}^{[s/o]}(k))$ (from now on denoted as $\underline{\Theta}(k)$ for brevity) after model training and how it is composed of different local model parameters by looking at an LMSSN with $n_x = 3$ state variables.

We will start with the parameter matrix $\underline{\Theta}(k)$ of a SISO LMSSN where each state and output equation is modeled by a MISO LMN (model architecture from Fig. 3.3 a)). The first LMN has two LLMs ($n_{m_1} = 2$), the second LMN has three LLMs ($n_{m_2} = 3$), the third LMN has one LLM ($n_{m_3} = 1$), and the output equation that is modeled by the fourth LMN has two LLMs ($n_o = 2$). An illustration of the time-varying $\underline{\Theta}(k)$ can be seen in Fig. 3.10. At each time step, the matrix is built according to the current values of the validity functions and their corresponding parameter values of the LLMs. Every entry in $\underline{\Theta}(k)$ is a weighted sum of the corresponding parameters from the different local models. Take, for example, parameter $a_{1,2}(k)$. It is calculated as

$$a_{1,2}(k) = a_{1,2,1}\mathit{\Phi}_{1,1}^{[s]}(k) + a_{1,2,2}\mathit{\Phi}_{1,2}^{[s]}(k)\,, \tag{3.11}$$

where the partition of unity states

$$\mathit{\Phi}_{1,1}^{[s]}(k) + \mathit{\Phi}_{1,2}^{[s]}(k) = 1\,. \tag{3.12}$$

The same goes for the calculations of all other parameters of $\underline{\Theta}(k)$, with the difference that the number of LLMs may vary, depending on the state or output equation that the parameter lies in (depending on the row of the matrix). The feature that nonlinearities can be learned separately for each state variable (with individual partitionings of the inner input space $[\underline{\hat{x}}^T(k),\, u(k)]^T$) is unique to LMSSN and has, to the author's knowledge, not been done in any other system identification method before.

Next, we want to consider the model architecture d) from Fig. 3.3. Again, a SISO LMSSN with three state variables is considered, but this time all three state variables are modeled with a single MIMO LMN and the output equation is an affine function. The parameter matrix $\underline{\Theta}(k)$ can be seen in Fig. 3.11.

Lastly, an LMSSN with two inputs and one output ($n_p = 2$ and $n_q = 1$) is demonstrated in Fig. 3.12. This setup has been chosen to demonstrate the LMSSN's flexibility regarding the arrangement of LMNs (which is here a combination of setup a) and c) of Fig. 3.3 for the state equation) and the straightforward extension to multiple dynamical inputs. State variables $\hat{x}_1(k)$ and $\hat{x}_2(k)$ are together modeled by

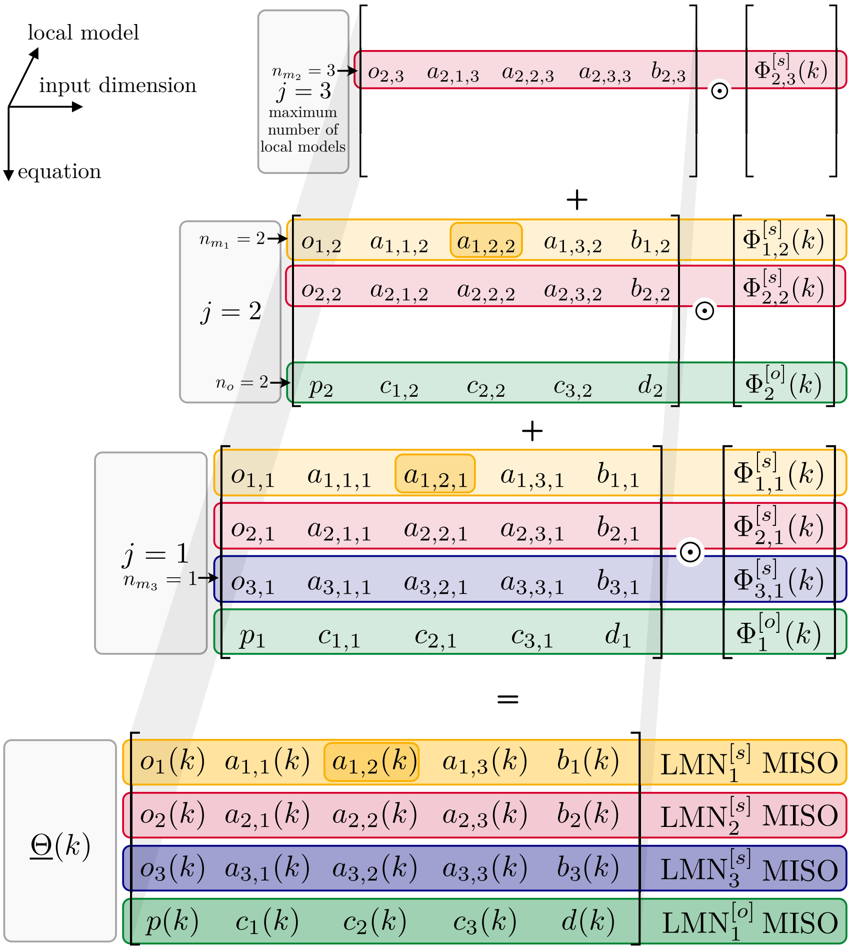

local model
input dimension
equation
$n_{m_2} = 3$
$j = 3$
maximum number of local models
$o_{2,3}$ $a_{2,1,3}$ $a_{2,2,3}$ $a_{2,3,3}$ $b_{2,3}$
$\odot$
$\Phi^{[s]}_{2,3}(k)$
$+$
$n_{m_1} = 2$
$j = 2$
$n_o = 2$
$o_{1,2}$ $a_{1,1,2}$ $a_{1,2,2}$ $a_{1,3,2}$ $b_{1,2}$
$o_{2,2}$ $a_{2,1,2}$ $a_{2,2,2}$ $a_{2,3,2}$ $b_{2,2}$
p_2 $c_{1,2}$ $c_{2,2}$ $c_{3,2}$ d_2
$\Phi^{[s]}_{1,2}(k)$
$\Phi^{[s]}_{2,2}(k)$
$\Phi^{[o]}_{2}(k)$
$+$
$j = 1$
$n_{m_3} = 1$
$o_{1,1}$ $a_{1,1,1}$ $a_{1,2,1}$ $a_{1,3,1}$ $b_{1,1}$
$o_{2,1}$ $a_{2,1,1}$ $a_{2,2,1}$ $a_{2,3,1}$ $b_{2,1}$
$o_{3,1}$ $a_{3,1,1}$ $a_{3,2,1}$ $a_{3,3,1}$ $b_{3,1}$
p_1 $c_{1,1}$ $c_{2,1}$ $c_{3,1}$ d_1
$\Phi^{[s]}_{1,1}(k)$
$\Phi^{[s]}_{2,1}(k)$
$\Phi^{[s]}_{3,1}(k)$
$\Phi^{[o]}_{1}(k)$
$=$
$\underline{\Theta}(k)$
$o_1(k)$ $a_{1,1}(k)$ $a_{1,2}(k)$ $a_{1,3}(k)$ $b_1(k)$
$o_2(k)$ $a_{2,1}(k)$ $a_{2,2}(k)$ $a_{2,3}(k)$ $b_2(k)$
$o_3(k)$ $a_{3,1}(k)$ $a_{3,2}(k)$ $a_{3,3}(k)$ $b_3(k)$
$p(k)$ $c_1(k)$ $c_2(k)$ $c_3(k)$ $d(k)$
LMN$^{[s]}_1$ MISO
LMN$^{[s]}_2$ MISO
LMN$^{[s]}_3$ MISO
LMN$^{[o]}_1$ MISO

Figure 3.10: Composition of time-varying matrix $\underline{\Theta}(k)$ for model architecture a) from Fig. 3.3. The three upper matrices are multiplied by the validity functions to the right (all entries in one row are multiplied element-wise with the validity function in the same row denoted by the Hadamard product $\odot$) and then added up to form $\underline{\Theta}(k)$ (matrix at the bottom). Per matrix, all parameters in one row belong to the same state or output equation and parameters in the same column belong to the same inner input dimension. The *third dimension* indicates to which LLM the parameter belongs.

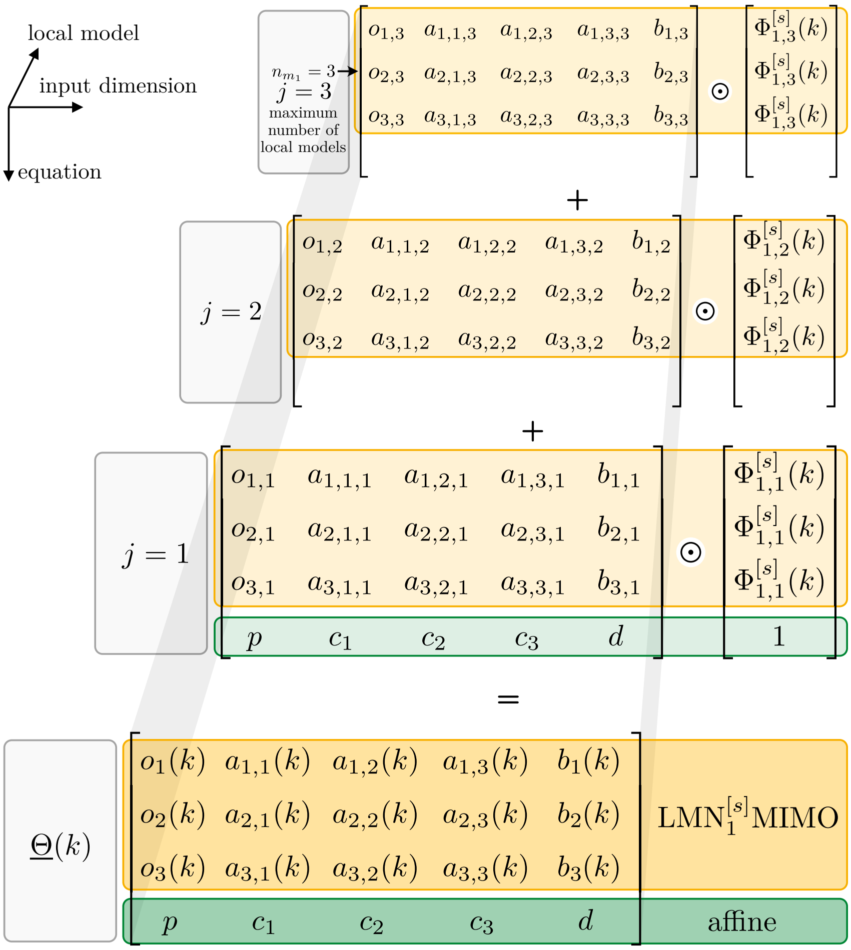

local model
input dimension
equation
$n_{m_1} = 3$
$j = 3$
maximum number of local models
$o_{1,3}$ $a_{1,1,3}$ $a_{1,2,3}$ $a_{1,3,3}$ $b_{1,3}$
$o_{2,3}$ $a_{2,1,3}$ $a_{2,2,3}$ $a_{2,3,3}$ $b_{2,3}$
$o_{3,3}$ $a_{3,1,3}$ $a_{3,2,3}$ $a_{3,3,3}$ $b_{3,3}$
$\odot$
$\Phi^{[s]}_{1,3}(k)$
+
$j = 2$
$o_{1,2}$ $a_{1,1,2}$ $a_{1,2,2}$ $a_{1,3,2}$ $b_{1,2}$
$o_{2,2}$ $a_{2,1,2}$ $a_{2,2,2}$ $a_{2,3,2}$ $b_{2,2}$
$o_{3,2}$ $a_{3,1,2}$ $a_{3,2,2}$ $a_{3,3,2}$ $b_{3,2}$
$\odot$
$\Phi^{[s]}_{1,2}(k)$
+
$j = 1$
$o_{1,1}$ $a_{1,1,1}$ $a_{1,2,1}$ $a_{1,3,1}$ $b_{1,1}$
$o_{2,1}$ $a_{2,1,1}$ $a_{2,2,1}$ $a_{2,3,1}$ $b_{2,1}$
$o_{3,1}$ $a_{3,1,1}$ $a_{3,2,1}$ $a_{3,3,1}$ $b_{3,1}$
p c_1 c_2 c_3 d
$\odot$
$\Phi^{[s]}_{1,1}(k)$
1
=
$\underline{\Theta}(k)$
$o_1(k)$ $a_{1,1}(k)$ $a_{1,2}(k)$ $a_{1,3}(k)$ $b_1(k)$
$o_2(k)$ $a_{2,1}(k)$ $a_{2,2}(k)$ $a_{2,3}(k)$ $b_2(k)$
$o_3(k)$ $a_{3,1}(k)$ $a_{3,2}(k)$ $a_{3,3}(k)$ $b_3(k)$
p c_1 c_2 c_3 d
$\mathrm{LMN}^{[s]}_1\mathrm{MIMO}$
affine

Figure 3.11: Composition of time-varying matrix $\underline{\Theta}(k)$ for model architecture d) of Fig. 3.3. The three state equations are coupled as there is only one validity function for all of them. There are no LLMs for the output equation as it is an affine function.

LMN$_1$ (MIMO LMN), which has $n_{m_1} = 3$ LLMs. LMN$_2$ has $n_{m_2} = 1$ LLM, which models state variable $\hat{x}_3(k)$ (MISO LMN). The output equation is also a MISO LMN with $n_o = 2$ LLMs. Note that the difference between a multiple-input and a single-input LMSSN is the existence of the last column in $\underline{\Theta}(k)$, which corresponds to the parameters for the second dynamical input. Likewise, the extension from the single-output LMSSN to the multiple-output case would only require the expansion of $\underline{\Theta}(k)$ by one additional row.

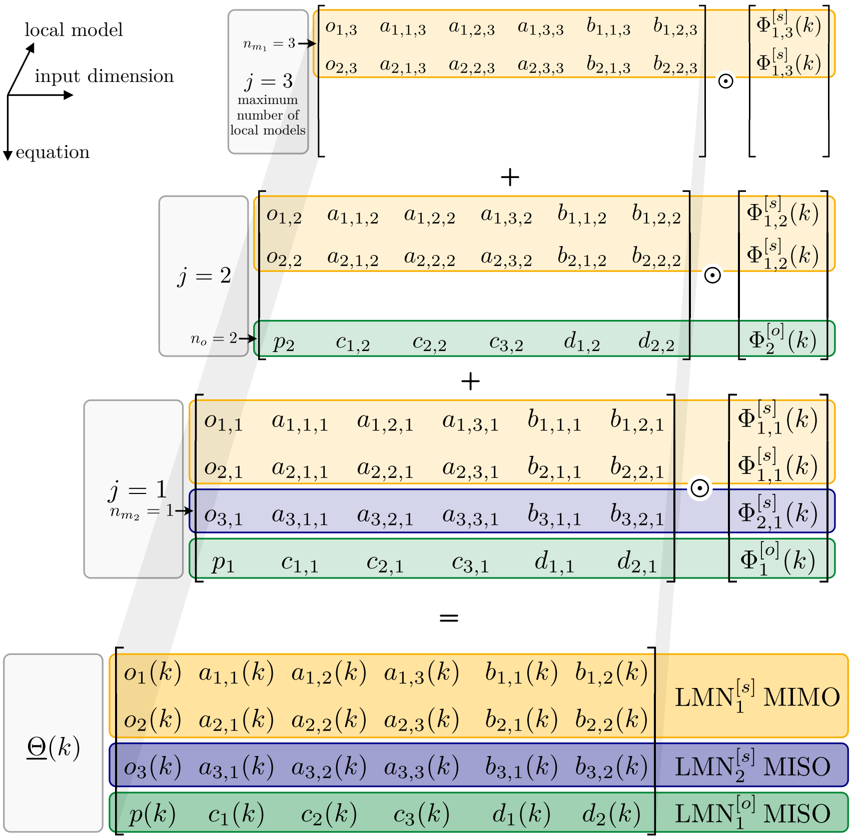

local model
input dimension
equation
$n_{m_1} = 3$
$j = 3$
maximum number of local models
$o_{1,3}$ $a_{1,1,3}$ $a_{1,2,3}$ $a_{1,3,3}$ $b_{1,1,3}$ $b_{1,2,3}$
$o_{2,3}$ $a_{2,1,3}$ $a_{2,2,3}$ $a_{2,3,3}$ $b_{2,1,3}$ $b_{2,2,3}$
$\odot$
$\Phi^{[s]}_{1,3}(k)$
$\Phi^{[s]}_{1,3}(k)$
+
$j = 2$
$n_o = 2$
$o_{1,2}$ $a_{1,1,2}$ $a_{1,2,2}$ $a_{1,3,2}$ $b_{1,1,2}$ $b_{1,2,2}$
$o_{2,2}$ $a_{2,1,2}$ $a_{2,2,2}$ $a_{2,3,2}$ $b_{2,1,2}$ $b_{2,2,2}$
p_2 $c_{1,2}$ $c_{2,2}$ $c_{3,2}$ $d_{1,2}$ $d_{2,2}$
$\odot$
$\Phi^{[s]}_{1,2}(k)$
$\Phi^{[s]}_{1,2}(k)$
$\Phi^{[o]}_{2}(k)$
+
$j = 1$
$n_{m_2} = 1$
$o_{1,1}$ $a_{1,1,1}$ $a_{1,2,1}$ $a_{1,3,1}$ $b_{1,1,1}$ $b_{1,2,1}$
$o_{2,1}$ $a_{2,1,1}$ $a_{2,2,1}$ $a_{2,3,1}$ $b_{2,1,1}$ $b_{2,2,1}$
$o_{3,1}$ $a_{3,1,1}$ $a_{3,2,1}$ $a_{3,3,1}$ $b_{3,1,1}$ $b_{3,2,1}$
p_1 $c_{1,1}$ $c_{2,1}$ $c_{3,1}$ $d_{1,1}$ $d_{2,1}$
$\odot$
$\Phi^{[s]}_{1,1}(k)$
$\Phi^{[s]}_{1,1}(k)$
$\Phi^{[s]}_{2,1}(k)$
$\Phi^{[o]}_{1}(k)$
=
$\underline{\Theta}(k)$
$o_1(k)$ $a_{1,1}(k)$ $a_{1,2}(k)$ $a_{1,3}(k)$ $b_{1,1}(k)$ $b_{1,2}(k)$
$o_2(k)$ $a_{2,1}(k)$ $a_{2,2}(k)$ $a_{2,3}(k)$ $b_{2,1}(k)$ $b_{2,2}(k)$
$o_3(k)$ $a_{3,1}(k)$ $a_{3,2}(k)$ $a_{3,3}(k)$ $b_{3,1}(k)$ $b_{3,2}(k)$
$p(k)$ $c_1(k)$ $c_2(k)$ $c_3(k)$ $d_1(k)$ $d_2(k)$
LMN$^{[s]}_1$ MIMO
LMN$^{[s]}_2$ MISO
LMN$^{[o]}_1$ MISO

Figure 3.12: Composition of time-varying matrix $\underline{\Theta}(k)$ for a MISO LMSSN with a mix of MIMO and MISO LMNs.

3.2 Construction Algorithms

In the previous section, we analyzed the LMSSN model structure without any insight into how this structure comes about. One surely could simply *guess* or make a physics-informed structure assumption and then learn the parameters of this model structure. A more sophisticated and general strategy that we want to employ here is the use of tree-construction algorithms, which estimate the structure in an incremental manner purely from data.

3.2.1 Local Linear Model Tree

The local linear model tree (LOLIMOT) [90] is a tree-construction algorithm that partitions the input space (in our case $\underline{\tilde{u}}(k) = [\underline{\hat{x}}^T(k), u(k)]^T$) by axis-orthogonal splits. Each resulting region of the input space is then modeled by a unique LM. The splits are performed by heuristic placement of the centers and standard deviations of local radial basis functions (RBFs) (see Sect. 2.2.2) as they serve in their normalized form as validity functions for the LMs. Each iteration, the input space of the LMN is split once and one further LM is added, increasing the capability of the network to capture nonlinearities.

Before the start of the tree construction, the input space is normalized to an operating regime between 0 and 1 for all dimensions of the input space. For the whole tree-construction algorithm to work, only one 'fiddle' parameter has to be specified a priori by the user. That is the proportionality factor between the operating regime width and the standard deviation of the RBFs. The default value is chosen as

$$k_\sigma = 0.4\,, \tag{3.13}$$

as is recommended in [89]. The standard deviation is thus calculated as

$$\underline{\sigma}_j = k_\sigma \cdot \underline{\Delta}_j\,, \tag{3.14}$$

where $\underline{\Delta}_j = [\Delta_{j1},\, \Delta_{j2},\, \ldots, \Delta_{j(n_x+1)}]^T$ includes the widths of the region of validity of LM j in all input dimensions $l = 1, \ldots, (n_x + 1)$. In the LOLIMOT case, the nonlinear parameters of the validity function in (3.2) are thus the centers $\underline{\mu}_j$ and standard deviations $\underline{\sigma}_j$ of the RBF belonging to the j-th LM $\underline{\theta}_j^{[nl]} = \left[\underline{\mu}_j^T,\, \underline{\sigma}_j^T\right]^T$.

The LOLIMOT algorithm adapted to construct the LMSSN model can be summarized as follows:

1. *Start with an initial model*: Initialize the LMSSN with the best linear approximation (BLA) [78] (see Sect. 2.1.2.2). Set n_{m_i} for all $i = 1, \ldots, n_x$ (for MISO state equations) or n_m (for MIMO state equations) to 1 and for the MISO output equation $n_o = 1$. This means that one LM is initialized for each state and output equation.

2. *Affine transformation of the state space*: Scale the state trajectory into the unit hypercube (see Sect. 3.3.3) which is necessary to properly place splits in the inner input space in Step 4. The affine transformation leads to a global affine model.

3. *Find the worst LM (or LMs) over all state and output equations*: Calculate a local loss function for each LM in each LMN. For the MISO output equation,

the local loss function can be computed by weighting the squared errors with the validity of the corresponding LM according to

$$I_m^{[o]} = \sum_{k=1}^{N} e^2(k) \Phi_m^{[o]}(k) \,. \tag{3.15}$$

For the state equation LMs, the error needs to be back-propagated, which is achieved by weighting with the absolute values of the parameters $c_{i,m}$. This leads to the loss function

$$I_{i,j}^{[s]} = \sum_{k=1}^{N} e^2(k) \Phi_{i,j}^{[s]}(k) \sum_{m=1}^{n_o} \Phi_m^{[o]}(k) |c_{i,m}| \tag{3.16}$$

for the MISO output equation and simplifies to weighting with $|c_i|$ in case of the affine output equation. Now, the worst-performing LM can be determined by considering the LM with the highest loss function value. It can be chosen as (i) over all LMNs only one LM is selected, or (ii) for each LMN the worst LM is selected. When enough computational power is available, also (iii) the n worst LMs may be selected.

4. *Check all splits*: The selected LMs from Step 3 are split by axis-orthogonal splits. Divisions in all dimensions are compared. For all $n_x + 1$ dimensions, the following steps are carried out:

   a) Determination of center coordinates and standard deviations for both newly established (hyper)rectangles

   b) Construction of all validity functions

   c) Nonlinear optimization of all model parameters

   d) Affine transformation of the state space (see Sect. 3.3.3)

5. *Find best split*: The best (the one with the lowest normalized root mean squared error (NRMSE) (5.2) on training data) of the split models from Step 4 becomes the refined model.

6. *Test for convergence*: The algorithm terminates if the model error of the current split is worse than the error of the penultimate split on validation data. Alternatively, a certain number of splits or the corrected Akaike information criterion (AIC$_\mathrm{c}$) may be used as termination criteria. If the criterion is not met, the algorithm starts over at Step 3.

Figure 3.13 illustrates the algorithm's operation for $n_x = 1$ in the first five iterations for one MISO state equation and an affine output equation. In this case, only one LMN is constructed in the input space $[\tilde{u}_1\, \tilde{u}_2] = [\hat{x}\, u]$. In the first iteration, the global model is estimated. The input space is then divided by an axis-orthogonal split, leading to two different split options. For both models, the parameters are estimated. The better-performing model is selected for further splitting. The model with the highest local error value is selected to be split further. This procedure is continued until the desired model complexity is reached.

3.2.2 Hierarchical Local Model Tree

The hierarchical local model tree (HILOMOT) algorithm [88, 89] works in principle quite similar to LOLIMOT. Instead of ensuring the partition of unity by normalization of RBFs, a hierarchical structure of sigmoid functions is employed (see Sect. 2.2.2). The differences occurs in Step 4 of the identification procedure:

1. - 3. Equivalent to LOLIMOT

4. *Check all splits*: The selected LMs are split by an axis-oblique split. The splits are initialized with $n_x + 1$ different axis-orthogonal splits and with the oblique split direction of the parent model (one hierarchy level up):

   a) Determination of center coordinates of the parent local model by calculating the center of gravity of all data points weighted in their respective region of validity as

   $$\underline{\mu}_j = \sum_{k=1}^{N} \underline{\tilde{u}}(k) \cdot \frac{\Phi_j(\underline{\tilde{u}}(k))}{\sum_{k=1}^{N} \Phi_j(\underline{\tilde{u}}(k))} \,. \tag{3.17}$$

   b) Construction of all validity functions. The split is initialized so that it passes through the center of gravity $\underline{\mu}_j$ and is either orthogonal to the split dimension or in the same oblique direction as the parent model.

   c) Nonlinear optimization of all model parameters, i.e., $\underline{\Theta}$ and the split direction and position.

   d) Affine transformation of the state space (see Sect. 3.3.3).

5. - 6. Equivalent to LOLIMOT

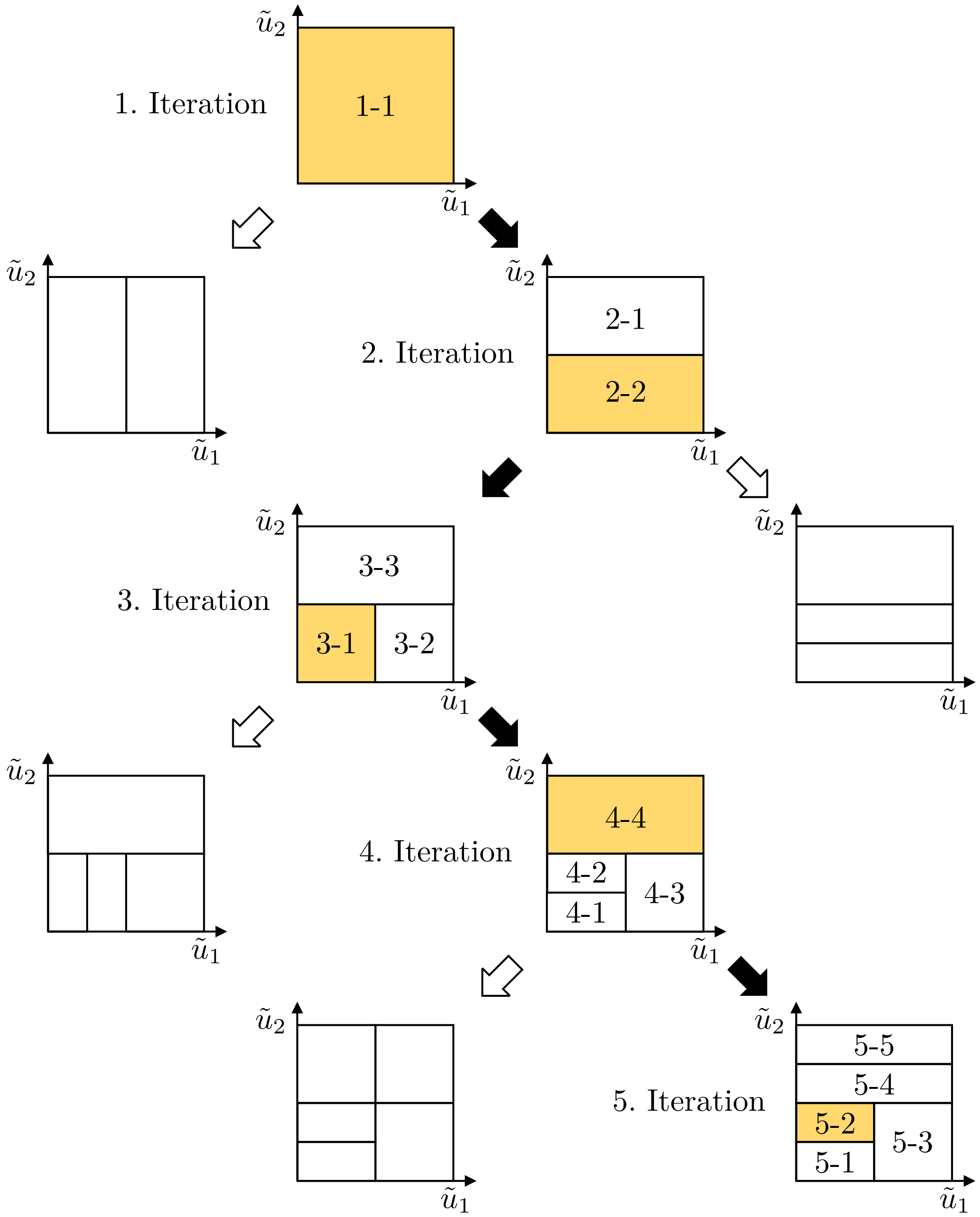


Figure 3.13: Demonstration of the LOLIMOT algorithm (see also [89]). In each iteration, the worst local model of the best performing model (yellow boxes) are split by axis-orthogonal splits. The final model is obtained when a termination criterion is reached.

How the hierarchy is built can be seen in Fig. 3.14. Here, one LMN is shown, which could also be the MISO LMN for the state equation of a first-order state space model with $[\tilde{u}_1\,\tilde{u}_2] = [\hat{x}\,u]$. In the first iteration, the hierarchy is initialized with a single validity function, which is valid in the whole input space. Since there is only one LM, this is to be split. Splits are initialized with axis-orthogonal splits through the center of gravity (3.17). After optimization of the parameters, the worst-performing LM is again split. This time, splits are initialized with axis-orthogonal splits through the center of gravity and the same split direction as of the parent node. For more details on the validity function construction, compare to Fig. 2.17.

3.3 Pivotal Steps in LMSSN Identification

Applying tree-construction algorithms to nonlinear state space models seems straightforward, but care has to be taken at some pivotal points during identification. Therefore, the model initialization, optimization, and transformation of the state space shall be explained in more detail.

3.3.1 Model Initialization

For every nonlinear optimization problem, one needs good initial parameters if local search methods shall be employed. In the case of nonlinear system identification, it seems sensible to start with a linear dynamic model for the following reasons [129]:

- the performance is as least as good as the linear model, which is further improved by nonlinear components (valid for nonlinear models which add nonlinear components on top of the linear model)
- problems of local optima are likely to be smaller when good initial linear estimates are available
- one starts with a stable model, which would be troublesome to archive for random initialization.

Therefore, we will use the BLA (see Sect. 2.1.2.2) as the starting point for the LMSSN. It has been proven in different works that the BLA is a viable initialization choice for nonlinear system identification [118, 64, 117].

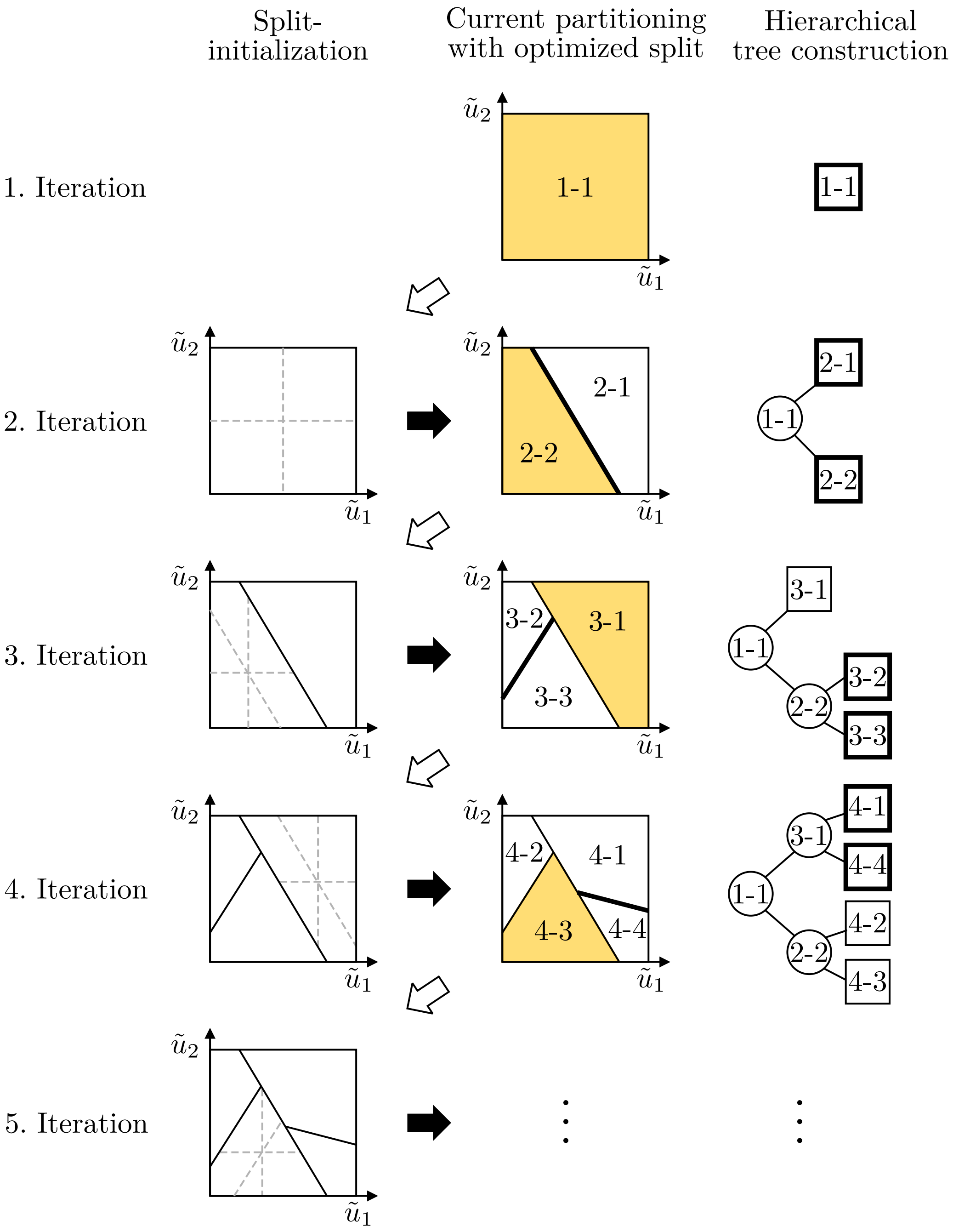


Figure 3.14: Demonstration of the HILOMOT algorithm (adapted from [47]). In each iteration, the worst local model of the best performing model (yellow boxes) are split. Splits are initialized with axis-orthogonal splits through the local model's center and with the parent node's split direction. The final model is obtained when a termination criterion is reached.

3.3.2 Optimization

The local model parameters of the state equation $\underline{\theta}^{[s]}$ and the parameters of the output equation $\underline{\theta}^{[o]}$ are optimized after each split. If normalized radial basis functions (NRBFs) are used and only axis-orthogonal splits are allowed, the parameters of the validity functions (center coordinates and standard deviations) do not need to be optimized as they are independently set by the LOLIMOT algorithm. For HILOMOT, when axis-oblique partitioning is wanted, besides the local model parameters, the split parameters $\underline{v}_j$ from (2.52) of all sigmoid functions (stored in $\underline{V}$) are optimized as well. Since usually for nonlinear black-box system identification no knowledge of the physical interpretation of the state vector is available, also no physics-informed initial values for the state vector $\underline{x}_0$ are available. Since all signals are scaled before identification in the interval 0 to 1, a reasonable choice for the initial state vector is $\underline{x}_0 = 0.5 \cdot \underline{1}^{n_x \times 1}$. This choice nevertheless leads to a transient phase, yielding low accuracy for the first time steps during optimization. This problem can be either overcome by neglecting or ignoring (weighting with 0) the transient phase during optimization[5] or by optimizing the initial state vector just like all other parameters (as was already explained in Sect. 2.3.4).

All optimizable parameters are finally concatenated in the vector

$$\underline{\theta} \; = \; [\underline{\theta}^{[s]^T} \quad \underline{\theta}^{[o]^T} \quad \underline{V}^T \quad \underline{x}_0^T]^T \,. \tag{3.18}$$

The analytical gradients with respect to all parameters of $\underline{\theta}$ can be found in Appx. B.3.

As default, the sum-of-squared-errors loss function

$$I(\underline{\theta}) = \sum_{k=1}^{N} e^2(k, \underline{\theta}) \tag{3.19}$$

is used. However, note that the generalization to other than Gaussian noise assumptions is easy and straightforward. The most prominent and important choice is to pursue robust statistics by replacing $e^2(k, \underline{\theta})$ with $|e(k, \underline{\theta})|$ in (3.19). This gives the LMSSN a flexibility advantage over least squares based methods because various error norms are possible.

[5] which might not be a problem for large datasets, but can be quite restricting, if data collection is expensive and not much data is generally available

Restricting Behavior in Certain Dimensions Different properties can be restricted within the tree construction in each LMN if additional knowledge about the process is available. Let the input space for the construction of the validity functions be denoted by $\underline{z}$ and the input space to the local models as $\underline{s}$. Both $\underline{z}$ and $\underline{s}$ are subsets of the inner input $\underline{\tilde{u}}$. This decomposition is well-known for local linear neuro-fuzzy models, where the $\underline{z}$ input space is called the rule premise input space and $\underline{s}$ the rule consequents input space [89], leading to the LMN equation

$$\hat{y} = \sum_{j=1}^{n_m} L_j(\underline{s}) \Phi_j(\underline{z}) \,. \tag{3.20}$$

Choosing different subsets of the inner input $\underline{\tilde{u}}$ for $\underline{s}$ and $\underline{z}$ leads to different model characteristics, as is depicted for a two-dimensional example in Fig. 3.15. Input dimension $\tilde{u}_1$ is for all cases a)-d) set to be splittable (included in $\underline{z}$) and local model weights in dimension $\tilde{u}_1$ can have different slopes (included in $\underline{s}$). The differences of sub-figures a) - d) and their title names are all regarding dimension $\tilde{u}_2$. In a), $\tilde{u}_2$ is not to be split and parameters are set to be 0 regarding this dimension leading to constant behavior in $\tilde{u}_2$. In b), $\tilde{u}_2$ is included in $\underline{s}$ so that all local model parameters regarding $\tilde{u}_2$ are equivalent. The way $\tilde{u}_2$ is included in $\underline{s}$ and $\underline{z}$ is equivalent in c), but local models are not forced to have the same slope in $\tilde{u}_2$. For any fixed value for $\tilde{u}_1$, the behavior is linear, but with different slopes for different operating points in $\tilde{u}_1$. This behavior can be compared to the behavior of a bilinear or linear parameter varying (LPV) function, where the scheduling is done in dimension $\tilde{u}_1$. The fully nonlinear case is shown in d). Here, $\tilde{u}_2$ is included in both $\underline{s}$ and $\underline{z}$. A case that is not shown here but could also be constructed is a switching system. In this case, $\tilde{u}_2$ would be included in $\underline{z}$ but not in $\underline{s}$. Dimension $\tilde{u}_2$ would then solely be used for scheduling.

Imposing this kind of structure on different parts of the LMSSN can lead to other well-known model structures such as block-oriented models. Some connections are drawn in Sect. 3.5.2.

Dealing with the Non-uniqueness of Nonlinear State Space Models Due to the similarity transformation (see Sect. 2.1.2.3), the nonlinear state space model has no unique representation. The transformation of the state as $\underline{\tilde{x}}(k) = \underline{T}^{-1}\underline{\hat{x}}(k)$ leaves the input-output behavior unaffected for infinitely many $\underline{T}$. The n_x^2 elements of $\underline{T}$ can be chosen freely under the condition that $\underline{T}$ is non-singular [96]. This leads to the problem for gradient-based optimization that n_x^2 redundant parameters are optimized,

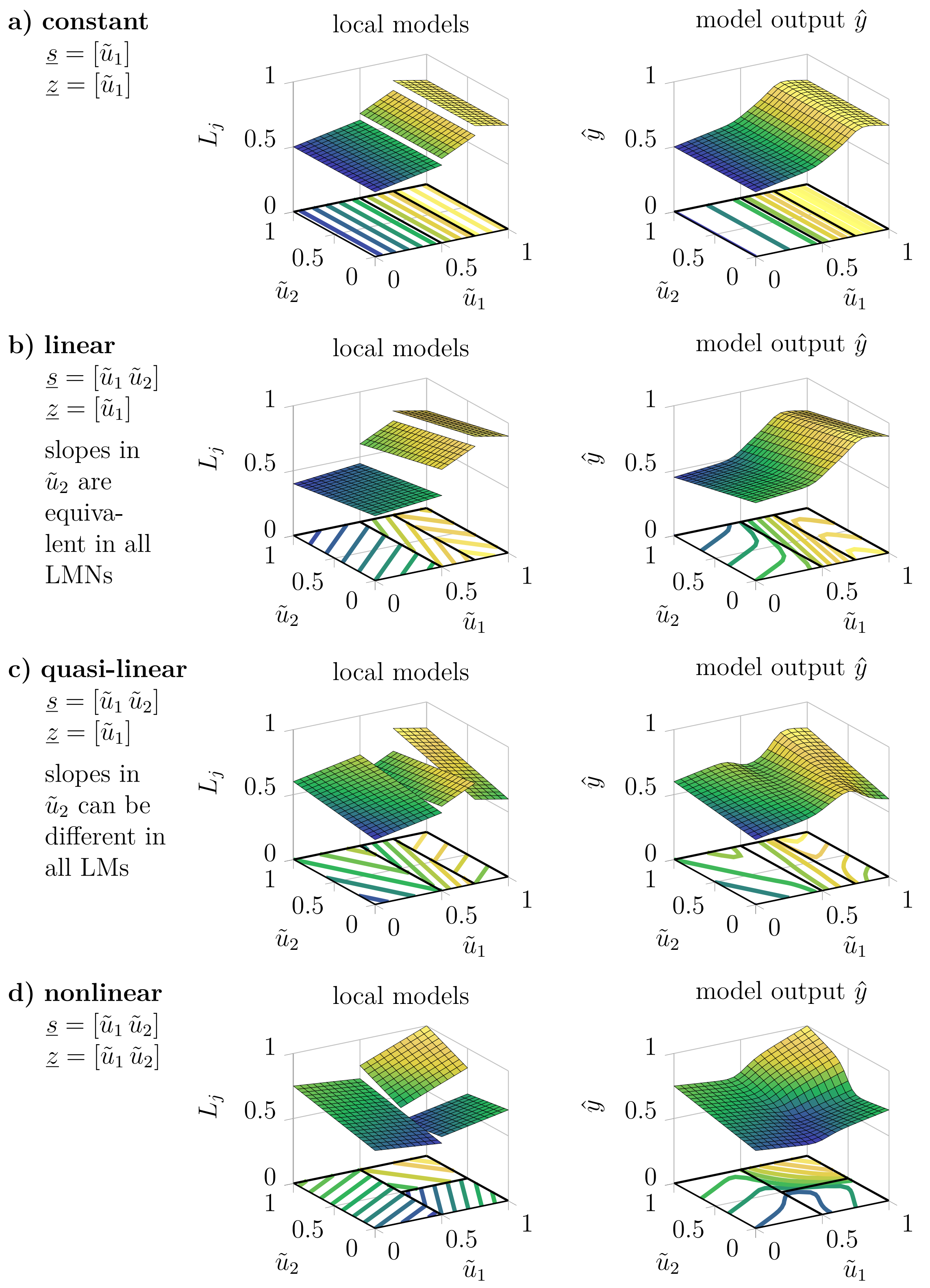


Figure 3.15: Different options to restrict the behavior of an LMN along certain dimensions. In the example, it is allowed to split in dimension $\tilde{u}_1$ and parameters in this dimension can vary from LM to LM (nonlinear behavior). The behavior for $\tilde{u}_2$ varies from a)-d) in the following way: a) constant and no split permissible, b) linear and no splits permissible, c) quasi-linear (local models do not share same weight for $\tilde{u}_2$, but global behavior is still linear) and no splits permissible, d) nonlinear and splits permissible.

leading to infinitely many solutions with identical input-output behavior.

One way to cope with this problem is by choosing canonical parameterizations (or their nonlinear counterparts). Unfortunately, it has been shown that those forms may lead to numerically ill-conditioned estimation problems [79].

Another way to deal with the non-uniqueness is by the use of data-driven local coordinates (DDLC) [79, 139]. The idea is to perform the optimization in a subspace of the original parameter space, in which all redundancies disappear. This is done by finding the class of all equivalent state space realizations and then constructing the space, which is locally orthogonal to the tangent space of all equivalent state space models. An illustration of how the method works on a linear first-order state space model in principle is shown in Fig. 3.16. The method boils down to finding a projection matrix $\underline{P} \in \mathbb{R}^{n_\theta \times (n_\theta - n_x^2)}$ such that the Jacobian of the loss function is calculated by

$$\underline{J}_{\text{DDLC}}(\underline{\theta}) = \underline{J}(\underline{\theta})\underline{P}\,, \tag{3.21}$$

with n_x^2 columns less than the original $\underline{J}(\underline{\theta})$ and full rank [96].

A third alternative to deal with the non-uniqueness is the use of a truncated singular value decomposition (SVD), for which it has been shown that it is equivalent to the

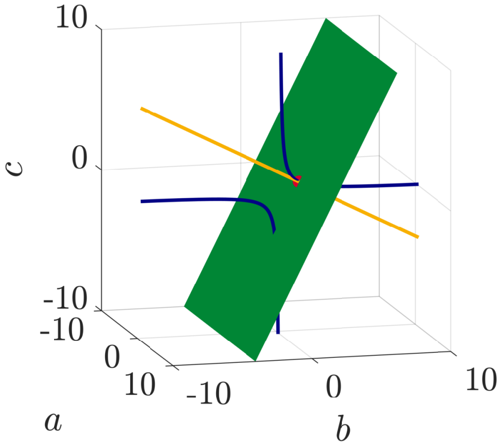


Figure 3.16: Construction of data-driven local coordinates of a linear first-order state space model with $(a, b, c, d) = (0.9, 1.4, \frac{1}{1.4}, 0)$, indicated by the red dot. The blue lines show equivalent state space models by similarity transformation for different values of T (which is a scalar for a first-order system). The yellow line is the linearization of all equivalent state space models at the point (a, b, c, d). The green plane is orthogonal to the linearization of the equivalent state space models, which is the subspace in which optimization shall be done (adapted from [152]).

method of DDLC [145]. The advantage of the DDLC, though, is less computational effort due to the n_x^2 fewer columns of the Jacobian. The derivation of $\underline{P}$ is for nonlinear state space approaches quite involved [96], which is why the third alternative will be employed for the LMSSN. How the truncated SVD is implemented within the Levenberg-Marquardt algorithm is explained in Appx. A.4.

Case Study: Effectiveness of Truncated SVD and DDLC The effectiveness of the truncated SVD (which is equivalent to the DDLC method) shall be demonstrated in two examples. First, the second-order nonlinear difference equation

$$\begin{aligned} y(k) = &- 0.07289 \left[u(k-1) - 0.2y^2(k-1)\right] \\ &+ 0.09394 \left[u(k-2) - 0.2y^2(k-2)\right] \\ &+ 1.68364\, y(k-1) - 0.70469\, y(k-2) \end{aligned} \tag{3.22}$$

will be identified with an LMSSN of second order. Training and validation data each have $N = 1024$ data samples with an signal-to-noise ratio (SNR) of 40 dB. The test dataset is undisturbed. For training, validation, and testing different realizations of an amplitude-modulated pseudo random binary signal (APRBS) are used. For nonlinear optimization, the Levenberg-Marquardt algorithm is run with and without SVD truncation for 100 iterations. For this small example, both algorithms converge to the exact same solution (within a tolerance of the final parameter values equal to $1 \cdot 10^{-10}$) with the same execution time (deviation of 1.2 %). Within the first split, the truncated SVD discards between 3-5 singular values. This is consistent with the expected redundancy of $n_x^2 = 4$ parameters. In a second example, for the modeling of $\mathrm{NO_x}$ emissions (see Sect. 5.6) the global affine model was estimated with and without SVD truncation of a second-order LMSSN. The training data has $N = 20\,700$ data points. After running the Levenberg-Marquardt algorithm for 100 iterations, both global affine models produce approximately the same training error (difference below 0.1 %), but the algorithm with the truncated SVD converged 13 % faster. The truncated SVD discarded 3-5 singular values, as is expected for a second-order model ($n_x^2 = 4$ redundant parameters).

These two examples (and other carried out studies) indicate that a speed-up can be expected from truncation, but a significant performance improvement might not be observable in practice. An additional speed-up may be achieved by focusing on more efficient implementations, which have not been elaborated on in this work. It can also be expected that the speed-up through truncation fades with an increasing number of

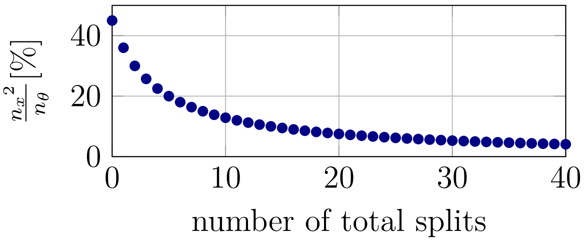


Figure 3.17: Percentage of redundant parameters in LMSSN SISO configuration for $n_x = 3$, $n_p = 1$, and $n_q = 1$ when the number of total splits increases (adapted from [152]).

parameters within the state space model as the share of redundant parameters with respect to all parameters decreases with increasing model complexity (see Fig. 3.17).

3.3.3 Transformation of State Space and Split Adaption

After each optimization step, the trajectory of the state vector changes within the state space. This is caused by the recurrent nature of the model, which leads to an altered distribution of data points in the state space. As the state space is part of the inner input space $\underline{\tilde{u}}$, the data point distribution in the input space changes during optimization. Such a problem does not occur for models, where the input space is spanned by current and previous process inputs (as for nonlinear finite impulse response (NFIR) models) and possibly outputs of the process (as for nonlinear autoregressive with exogenous input (NARX) models). Those data points do not change their location in regressor space while changing the model's parameters.

Exact Transformation For ease of notation for the transformation, we will use the following notation for the SISO LMSSN

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \sum_{j=1}^{\max(n_{m_i}, n_o)} \left(\underline{o}_j + \underline{A}_j \underline{\hat{x}}(k) + \underline{b}_j u(k)\right) \Phi_j^{[s]}(\underline{\hat{x}}(k), u(k)) \\ \hat{y}(k) &= \sum_{j=1}^{\max(n_{m_i}, n_o)} \left(p_j + \underline{c}_j^T \underline{\hat{x}}(k) + d_j u(k)\right) \Phi_j^{[o]}(\underline{\hat{x}}(k), u(k)) \,. \end{aligned} \tag{3.23}$$

The parameters $\underline{o}_j, \underline{A}_j, \underline{b}_j, p_j, \underline{c}_j^T, d_j$ stand for one "slice" in the *third* dimension of Fig. 3.10 as

$$\underline{\Theta}_j = \begin{bmatrix} \underline{o}_j & \underline{A}_j & \underline{b}_j \\ p_j & \underline{c}_j^T & d_j \end{bmatrix} . \tag{3.24}$$

The first "slice" $\underline{\Theta}_1$ is fully populated since we always start with an affine approximation. In all other "slices" ($j = 2, \ldots, \max(n_{m_i,n_o})$), some rows may contain parameters, while others do not (entries set to 0), depending on the way each individual state or output equation is split (e.g., Fig. 3.10 where for $j = 2$ the third row has no entries and for $j = 3$ entries are missing in all rows except for the second). Note that $\max(n_{m_i,n_o})$ is used for ease of notation without loss of generality as not all rows of each $\underline{\Theta}_j$ may contain parameters. This makes all combinations of MIMO and MISO LMNs and affine functions possible in state and output equation.

The state trajectory is transformed initially and after each optimization so that it fits exactly in the unit hypercube $[0, 1]^{n_x}$ (see Fig. 3.18). To accomplish this, for each state variable $\hat{x}_i$, an offset s_i^o and range parameter s_i^r is calculated according to

$$s_i^o = \min_k \hat{x}_i(k) \tag{3.25}$$

$$s_i^r = \max_k \hat{x}_i(k) - \min_k \hat{x}_i(k) \, . \tag{3.26}$$

Note that the minimum and the maximum within each state variable are found over the evolution of the state trajectory and thus over the discrete time k.

Now, the transformed state vector $\underline{\tilde{x}}(k)$ can be calculated by

$$\underline{\tilde{x}}(k) = \underline{T}^{-1}(\underline{\hat{x}}(k) - \underline{t}) \, , \tag{3.27}$$

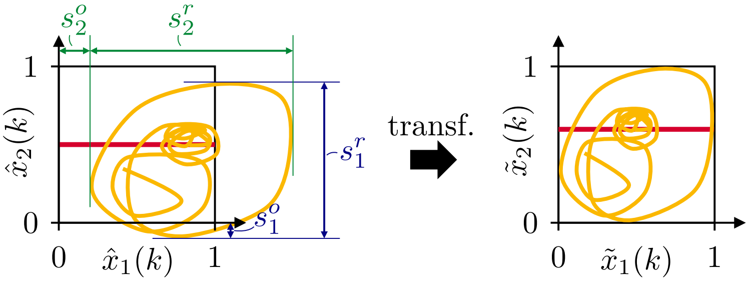


Figure 3.18: Transformation of state trajectory and split position for a two-dimensional example. The evolving state trajectory is transformed so that it fits inside the square $[0, 1]^2$ without changing the input-output behavior of the model.

where

$$\underline{T} = \begin{bmatrix} s_1^r & 0 & \cdots & 0 \\ 0 & s_2^r & & \\ \vdots & & \ddots & \vdots \\ 0 & & \cdots & s_{n_x}^r \end{bmatrix} \quad \text{and} \quad \underline{t} = \begin{bmatrix} s_1^o \\ s_2^o \\ \vdots \\ s_{n_x}^o \end{bmatrix} . \tag{3.28}$$

Using the relation (3.27) to transform (3.23) leads to

$$\begin{aligned} \underline{\tilde{x}}(k+1) &= \sum_{j=1}^{\max(n_{m_i},n_o)} \left(\underline{\tilde{o}}_j + \underline{\tilde{A}}_j \underline{\tilde{x}}(k) + \underline{\tilde{b}}_j u(k)\right) \Phi_j^{[s]} \left(\underline{T}\,\underline{\tilde{x}}(k) + \underline{t}, u(k)\right) \\ \hat{y}(k) &= \sum_{j=1}^{\max(n_{m_i},n_o)} \left(\tilde{p}_j + \underline{\tilde{c}}_j^T \underline{\tilde{x}}(k) + \tilde{d}_j u(k)\right) \Phi_j^{[o]} \left(\underline{T}\,\underline{\tilde{x}}(k) + \underline{t}, u(k)\right) , \end{aligned} \tag{3.29}$$

with the transformed scalars, vectors, and matrices

$$\begin{aligned} \underline{\tilde{o}}_j &= \underline{T}^{-1}\underline{o}_j + \underline{T}^{-1}\underline{A}_j\underline{T}\,\underline{t} - \underline{t} \\ \underline{\tilde{A}}_j &= \underline{T}^{-1}\underline{A}_j\underline{T} \\ \underline{\tilde{b}}_j &= \underline{T}^{-1}\underline{b}_j \\ \tilde{p}_j &= p_j + \underline{c}_j^T\underline{T}\,\underline{t} \\ \underline{\tilde{c}}_j^T &= \underline{c}_j^T\underline{T} \\ \tilde{d}_j &= d_j \, . \end{aligned} \tag{3.30}$$

The transformations of the validity functions (and thus the split location), initial condition, and transformation of a MISO LMSSN can be found in Appx. B.2.

It shall be noted that the transformed model (3.29) yields exactly the same input-output behavior as (3.23).

Alternative Approaches for State Trajectory Scaling In principle, other ways to ensure that the state trajectory resides in $[0,1]^{n_x}$ are possible. Either, (i) the unconstrained parameter optimization problem can be extended to a constrained optimization problem where violations of the element-wise condition $\underline{0} \leq \underline{\hat{x}}(k) \leq \underline{1}$ are penalized, e.g., by means of barrier functions. Alternatively, (ii) the state equation from (3.1) can be extended by a sigmoid function (applied element-wise) as

$$\underline{\hat{x}}(k+1) = \frac{1}{1 + e^{m(0.5\cdot\underline{1} - \underline{h}(\underline{\hat{x}}(k),u(k))}} \, . \tag{3.31}$$

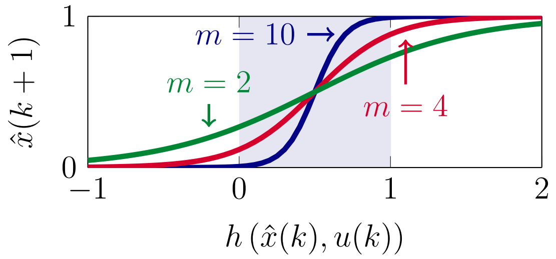


Figure 3.19: Sigmoid transformation. Parameter m adjusts the slope of the sigmoid and thus its sensitivity. The state trajectory should stay in the unit (hyper-)cube (▨).

This sigmoid is centered around $0.5 \cdot \underline{1}$ and the slope can be adjusted by parameter m, see Fig. 3.19.

Both alternative approaches make the optimization problem a lot harder to solve. In (i), this is the case through the introduction of constraints. In contrast, introducing a sigmoid nonlinearity in (ii) would only properly work for an autonomous system ($u(k) = 0$) as the influence of the process input moves the state vector into the saturation of the sigmoid and thus the vanishing gradient problem becomes dominant. Therefore, the exact affine transformation is preferred and employed throughout this thesis.

Transformation Embedded in Tree-Construction Algorithm How the optimization, transformation, and split in each iteration work is shown in Fig. 3.20 for the axis-orthogonal case[6]. The state space of different stages during the operation of the LMSSN algorithm is shown for a model with two states.

First Optimization When optimizing the initial model, the global affine model is estimated with the optimized parameters $\underline{\hat{\theta}}_{11}$, shown in the upper left of the figure. It can be seen that the trajectory of the state vector does not exclusively lie within the unit hypercube. Therefore, the model extrapolates in those regions where the trajectory leaves the square.

First Transformation In the first transformation, the state vector is adjusted so that extrapolation does not occur anymore. The parameters become $\underline{\tilde{\theta}}_{11}$. As no split has been done yet, the validity function stays the same (which is 1 for the global model for all data points in the input space anyway).

[6] Note that an extension to the axis-oblique case is straightforward and the implementation details can likewise be found in Appx. B.2.

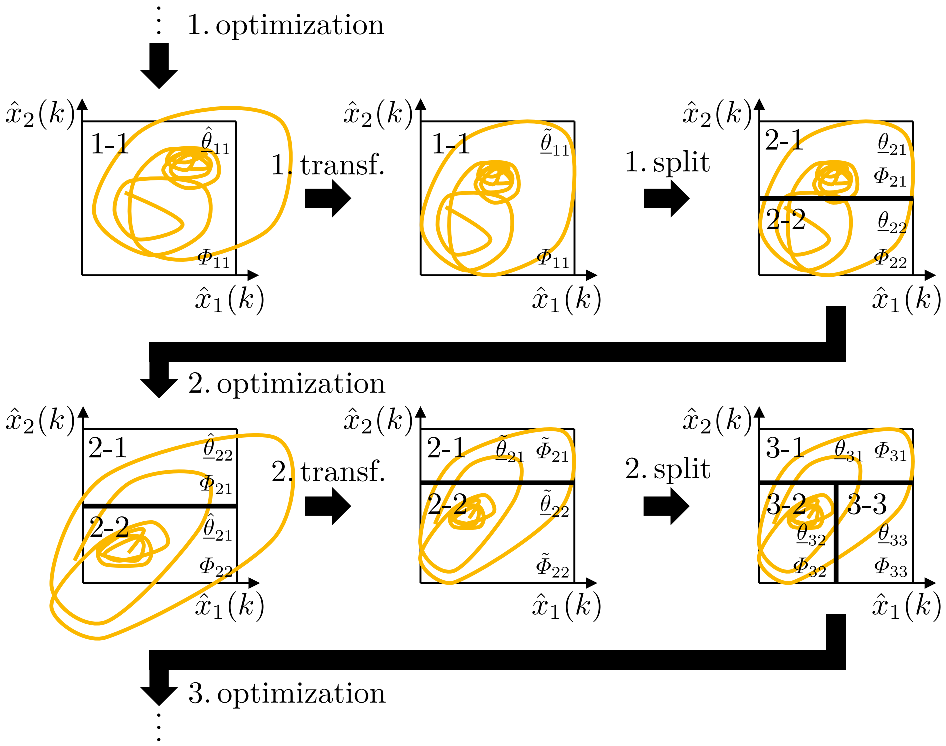


Figure 3.20: Transformation of state space and split adaption. The state trajectory changes its path in each optimization step within the state space. To make splitting possible, the state trajectory is rescaled after each optimization into the unit hypercube, which is in the interval $[0, 1]$ for each state dimension.

First Split After that, the global model is split the first time. All possible splits are carried out and the best split is chosen. The LM that is split (called the parent LM) inherits the optimized parameters to both new LMs (called the children LMs)

$$\underline{\theta}_{21} = \underline{\theta}_{22} \overset{!}{=} \tilde{\underline{\theta}}_{11} \,. \tag{3.32}$$

The children's center coordinates are inherited from the parent for the NRBF validity functions, except for the coordinate in the split dimension. In this dimension, the center coordinate is placed in the middle of the validity region of the newly established children LM. The same is done with the children's standard deviation: Its values are inherited from the parent LM, except for the split dimension. Here, the value is set by a user-defined proportionality factor k_σ. For hierarchical sigmoid functions, the center of gravity is estimated according to (3.17) and the initial split is axis-orthogonal to the split dimension.

Second Optimization The model with two LMs is now optimized. At this point, all possible splits of the worst LMs are tried and the nonlinear optimization is carried out for every model. The model with the lowest NRMSE (5.2) on training data becomes the new parent model.

Second Transformation Besides the transformation of the parameters ($\hat{\underline{\theta}}_{22}$ becoming $\tilde{\underline{\theta}}_{22}$), the split's position also needs to be transformed, leading to transformed validity functions. This is necessary to ensure that the data points still contribute to the same LM for which they were trained. If the LM still spans in a given dimension the whole input space (for axis-orthogonal splitting only), validity function parameters are not adjusted in this dimension to minimize unwanted normalization effects.

Second Split All unsplit LMs inherit all their properties, meaning their LM parameters and validity function parameters. The split LM (parent) passes the parameters and validity function parameters just like it was done in the first split.

From here on, every future iteration is carried out just like the second one until the termination criterion is reached.

3.4 Characteristics

Some properties and characteristics of the LMSSN will be highlighted. Among those are the universal approximator property, the extrapolation behavior, and stability assessment.

3.4.1 Universal Approximator Property

It has been shown by [58] that the multiple affine state space model

$$\begin{aligned} \hat{\underline{x}}(k+1) &= \sum_{j=1}^{n_m} \Big(\underline{o}_j + \underline{A}_j \, \hat{\underline{x}}(k) + \underline{b}_j \, u(k)\Big) \, \Phi_j(k) \\ \hat{y}(k) &= \sum_{j=1}^{n_m} \Big(p_j + \underline{c}_j^T \, \hat{\underline{x}}(k) + d_j \, u(k)\Big) \, \Phi_j(k) \end{aligned} \tag{3.33}$$

is a universal approximator to the nonlinear state space model (3.1). Note that the validity function $\Phi_j(k)$ is the same for the state and output equation. This model is a special case of the LMSSN when one MIMO LMN is used for state and output equations together. This makes the LMSSN a universal approximator as well.

3.4.2 Extrapolation

Extrapolation plays a crucial role in nonlinear system identification when the available training data does not cover the whole operating regime in which the model operates during testing. This scenario should be circumvented as far as possible, since generally speaking, it is never a good idea to extrapolate. There is only one exception to the rule, that is if there exists an exact match between the process and model structure [96]. Nevertheless, one can easily think of collected data for training, in which some crucial regions of operation were not captured. It is a serious issue if a model shows erratic behavior only because it has never encountered this operating region before.

In those cases in which extrapolation cannot be avoided, the question arises, which extrapolation behavior – constant, linear, polynomial, etc. – is the most desirable in black-box modeling (see Fig. 3.21).

In LMSSNs, affine functions approximate different partitions in the input space, leading therefore to linear extrapolation. This property seems reasonable for nonlinear dynamic models and suggests that the LMSSN is in extrapolation "well behaved". In contrast, other models use other function approximation methods such as polynomials (see Sect. 2.1.2.3), consequently leading to polynomial extrapolation behavior. If

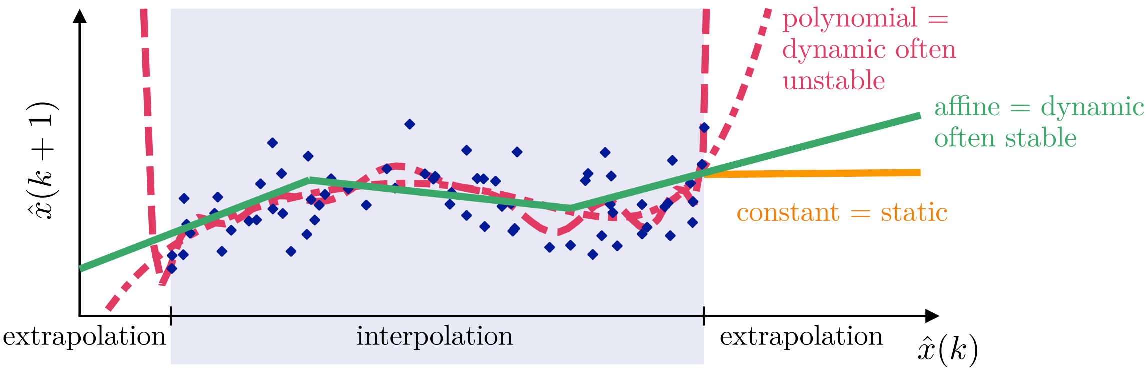


Figure 3.21: Illustration of different extrapolation behaviors (adapted from [89]) where the slope corresponds to the pole of the linearized system. (◆) measurements, (🟥) polynomials of order $n = 5$ and $n = 20$, (🟩) multiple affine models, (🟧) constant extrapolation.

the degree of the polynomial model exceeds a reasonably small number, the model's output tends to be erratic near the interpolation boundaries and tends to $+\infty$ or $-\infty$ very quickly, often yielding unstable behavior. Extrapolation can be, therefore, quite dangerous for polynomial models [50].

Case Study: Extrapolation The extrapolation behavior of the LMSSN (with NRBF and hierarchical sigmoid validity functions) will be investigated on a second-order Hammerstein test process. For the static nonlinearity, a quadratic function is used. For the linear system, a second-order oscillatory process with gain 1, damping 0.5, and a time constant of 5 sec is chosen. With a sampling time of $T_s = 1\,\text{sec}$, this process is described by the following nonlinear difference equation

$$\begin{aligned} y(k) = {} & 0.01867\, u^2(k-1) + 0.01746\, u^2(k-2) \\ & + 1.7826\, y(k-1) - 0.8187\, y(k-2)\,. \end{aligned} \tag{3.34}$$

Models are trained with $N = 2048$ data samples of an odd multisine with a maximum frequency of 0.4 Hz, while their model complexity is determined through a validation dataset, also with $N = 2048$ data points of an odd multisine signal. The mean of the signals is chosen to be 1, and the root mean squared (rms) value of the signal is also 1.[7] The training and validation output have output noise added with an SNR of 40 dB. The LMSSN algorithm terminates if the validation error of the current split is worse than the validation error of the previous to last split. Two LMSSN models are trained: the first with NRBF validity functions (LOLIMOT) and another with hierarchical sigmoid functions (HILOMOT) with the restriction that only axis-orthogonal splits are allowed. Therefore, both LMSSN models are similar in the way the algorithms work and only differ in the validity function construction.

In comparison, the polynomial nonlinear state space model (PNLSS) is examined (see Sect. 2.1.2.3). The MATLAB® toolbox PNLSS v1.0 [135] is used for training of the models. The Levenberg-Marquardt optimization is carried out for 100 iterations. Afterward, all iterations of the optimized model are compared on validation data and the best performing one is chosen as the final model (early stopping). The model order is chosen according to the order of the process, while for the monomials in the state and output equation, all different combinations with a degree up to 5 are evaluated. In total, 256 PNLSS models were evaluated. For example, a model with

[7] A mean-free training signal is not chosen to obtain a BLA that is unequal to 0, which would happen otherwise due to the quadratic nonlinearity.

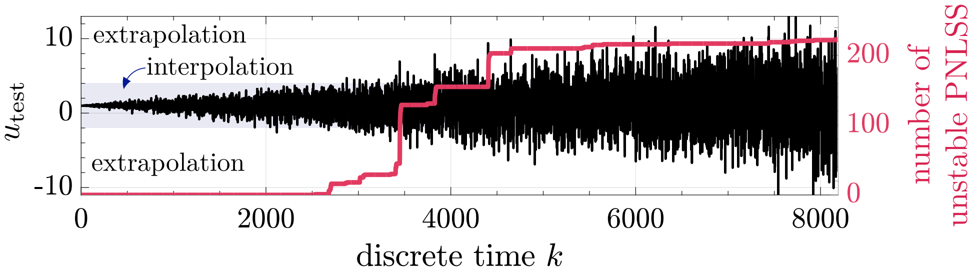


Figure 3.22: Multisine test signal with increasing maximum amplitude (left ordinate) and the number of unstable PNLSS models (right ordinate). For amplitudes greater than |3|, extrapolation occurs. Only 33 of 256 PNLSS models stay stable until the end of the increasing test signal.

monomial degrees 2 and 3 in the state equation and no polynomial extension in the output equation will be denoted by $\mathcal{O}(\underline{\zeta}) = [2\ 3]$ and $\mathcal{O}(\underline{\eta}) = [\ ]$. Note that degree one is always included in the linear model.

The severity of extrapolation with polynomial models is shown in Fig. 3.22. The shown test input signal is an odd multisine with a maximum frequency of 0.4 Hz with an increasing maximum amplitude up to an rms value of 4. The regions are indicated for which interpolation and extrapolation occurs, depending on the test signal's amplitude. The amplitude of the input signal is after round about 2100 data points the first time greater than |3| and therefore operates in extrapolation. The number of unstable PNLSS models is plotted from the right side of the figure. In total, 223 PNLSS models get unstable and only 33 were stable until the maximum amplitude of 13. Those are almost exclusively models that did not have any polynomials in the state equation or only second-order polynomials[8]. This means that, in general, only models with a linear state equation or a state equation that exactly fits the process stays stable. This dramatically limits the PNLSS's usefulness for black-box modeling, where the underlying process is unknown.

Next, the steady-state characteristic curve is analyzed. Figure 3.23 shows the comparison of the static behavior of the BLA and four selected PNLSS models with respect to the true processes static curve. The BLA is inherently unable to approximate a polynomial process. If the order in the state equation is chosen correctly and the output order is low, PNLSS is able to identify the process correctly (Fig. 3.23 far left). On the other hand, if higher order terms are chose as well, PNLSS produces only reasonable results within the interpolation boundaries and quite erratic behavior in extrapolation.

[8] One single model with third-order and fourth-order monomials in the state equation stayed stable.

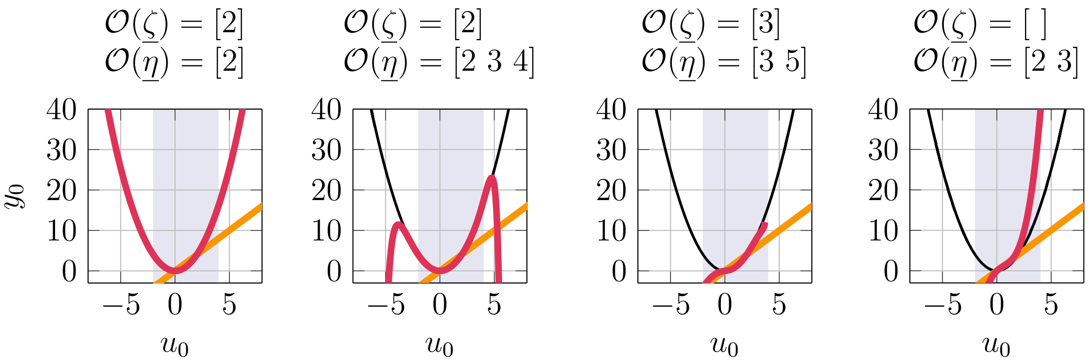


Figure 3.23: Comparison of static behavior of the (—) Hammerstein process with (▬) BLA and (▬) different PNLSS models of different degrees. The light blue area (▭) indicates the interpolation region. The end of the lines within the plot indicate the threshold to instability.

Both estimated LMSSN models have in total 17 LMs. The LMSSN models approximate the process well in interpolation but with increasing input amplitudes diverge continuously in extrapolation. In Fig. 3.24, the best models with one, three, nine, and 14 splits are shown. Numerical issues occurred for the LMSSN models with NRBF validity functions (too small values for the RBFs), indicated by the end of the LMSSN lines. This problem is further investigated in Sect. 3.6.1. Also, for the NRBF case, reactivation is present in a few cases (Fig. 3.24 upper two middle plots). This is due to different standard deviations of the RBFs in one dimension. Both of these problems are overcome by choosing hierarchical sigmoid validity functions, which produce qualitatively similar results without reactivation and numerical problems at the cost of higher computation times due to the more complicated validity function construction.

Concluding the analysis of the Hammerstein model, only the correct choice in model complexity leads to an accurate fit of the PNLSS model. In this case, the model works perfectly in interpolation and extrapolation. Extensive prior process knowledge is necessary to ensure that a good model fit is possible. The LMSSN can never match the true process in extrapolation as LMSSN extrapolates linearly, while the process is polynomial. Moreover, it extrapolates in a robust and reliable fashion.

Extrapolation Detection In cases when it cannot be ensured that extrapolation will not happen, it might be useful if the model is capable of at least *detecting* extrapolation. This information can be used to activate a base model in extrapolation, ensuring stability and enforcing a desired extrapolation behavior. Alternatively, a

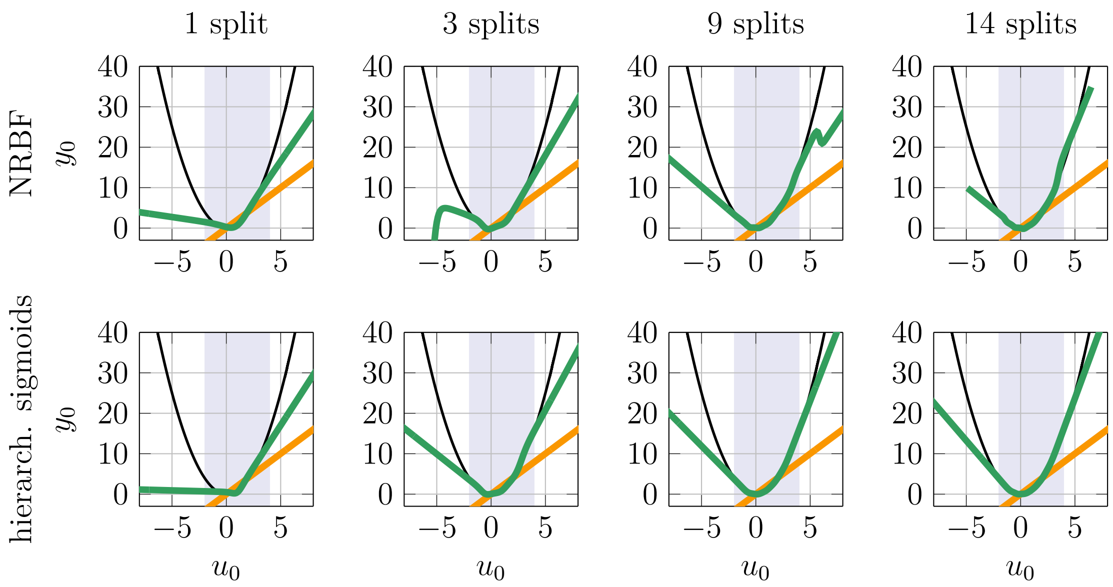


Figure 3.24: Comparison of static behavior of the (—) Hammerstein process with (—) BLA and (—) LMSSN models with different complexity. The upper and lower row show LMSSNs with NRBF and hierarchical sigmoid validity functions, respectively. The light blue area (▭) indicates the interpolation region. The end of the lines within the plot indicate the threshold to instability.

warning can be issued that at least it is known that the model operates in a dangerous operating regime.

For the above described second-order Hammerstein model, the state trajectory of the final LMSSN with hierarchical sigmoid validity functions is shown for a test input signal with increasing amplitude in Fig. 3.25. Here, the non-uniqueness of the state space model and the rescaling of the state trajectory (see Sect. 3.3.3) provide the LMSSN with the feature that extrapolation can be easily detected if the state trajectory leaves the interval $[0, 1]$ in any state dimension.

3.4.3 Stability

Unlike linear models, for nonlinear models, the concept of stability cannot be defined as a generic property of the model but has to be evaluated in the proximity of possibly multiple equilibria [2]. Since time recurrence only occurs in the state equation of a state space model, only the state equation has to be analyzed. An equilibrium $\underline{\hat{x}}_E$ for the state equation

$$\underline{\hat{x}}(k+1) = \underline{h}(\underline{\hat{x}}(k), u(k)) \tag{3.35}$$

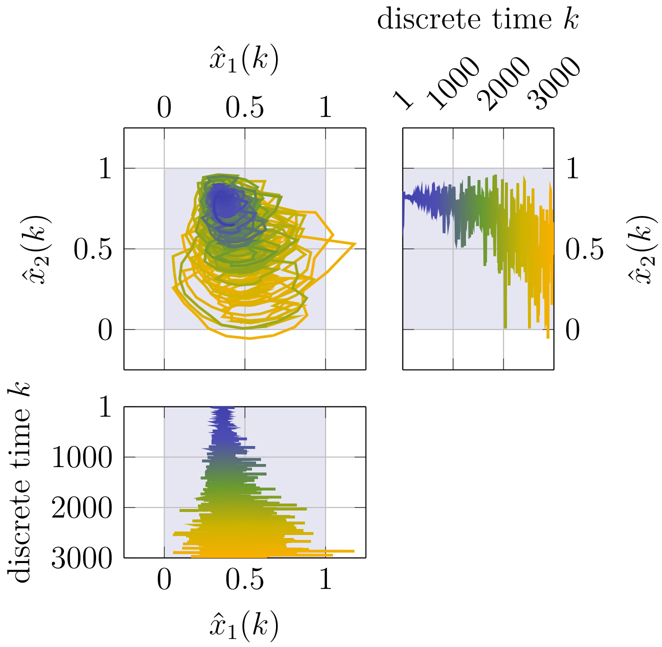


Figure 3.25: Extrapolation detection for the LMSSN model on the second-order Hammerstein process. As soon as a state variable operates outside the interval [0, 1] (light blue areas), extrapolation is detected. For this example, this happens the first time after roughly $k = 2900$ time steps. The changing color from blue to yellow indicates the progress in time.

is found as the point to which the model converges with a constant input signal u as

$$\underline{\hat{x}}_E = \underline{h}(\underline{\hat{x}}_E, u)\,. \tag{3.36}$$

An equilibrium is called stable, loosely speaking, if all trajectories $\underline{\hat{x}}(k)$ for $k = 0, \ldots, N$, that start in the neighborhood of $\underline{\hat{x}}_E$, converge with increasing time eventually to this equilibrium. The term stability is also used in a weakened form when the trajectory does not converge to the equilibrium but stays in a close neighborhood.

For nonlinear systems, commonly, Lyapunov stability [73] of autonomous systems (meaning that the input $u(k) = 0$) is used. Considering (3.35) with an equilibrium $\underline{\hat{x}}_E = \underline{0}$, the equilibrium is called *stable* in the sense of Lyapunov, if for every $\epsilon > 0$ there exists a $\delta > 0$ such that, if $||\underline{\hat{x}}(0)|| < \delta$, then for every $k \geq 0$ we have $||\underline{\hat{x}}(k) < \epsilon||$. If it also holds that

$$\lim_{k\to\infty} \underline{\hat{x}}(k) = \underline{0}\,, \tag{3.37}$$

then the equilibrium is called *asymptotically stable*.

Guaranteeing the stability of nonlinear dynamical systems in this way is not a straightforward task. Especially when the validity functions $\underline{\Phi}^{[s]}$ depend on the state vector $\underline{\hat{x}}(k)$ (which is the default for the LMSSN), results are hard to derive [139].

Lyapunov's direct method has been extended to local linear neuro-fuzzy models [134] as

$$\underline{A}_j^T\,\underline{P}\,\underline{A}_j - \underline{P} \prec 0 \qquad \text{for } j = 1, 2, \ldots, n_m\,, \tag{3.38}$$

with matrices $\underline{A}_j$ being the system matrices of the LLMs and $\underline{P}$ being positive definite. If a matrix $\underline{P}$ is found that satisfies (3.38) for all LLMs simultaneously, the model under test is guaranteed to be stable. This result, though, only holds for local *linear* and not local *affine* models. For local affine models, the equilibrium is not necessarily at $\underline{0}$, but multiple equilibria may exist. Also, (3.38) is only a sufficient condition, which is known to be quite conservative [154]. If the LMSSN is set up with a MIMO state equation and offsets are not used, this method is a way to check or ensure stability.

Limit Cycles For nonlinear systems, steady oscillation may occur for constant input signals u. The state vector repeats periodically for such oscillations and the state trajectory is a closed curve. For the state vector sequence

$$\underline{\hat{x}}(0), \underline{\hat{x}}(1), \ldots, \underline{\hat{x}}(l)\,, \tag{3.39}$$

it therefore occurs that

$$\underline{\hat{x}}(0) = \underline{\hat{x}}(l)\,. \tag{3.40}$$

The limit cycle contains $l + 1$ steps. Therefore, one speaks of a limit cycle of length $l + 1$.

Case Study: Settling Behavior of LMSSN The LMSSN uses in its internal calculations scaled versions of the inputs, outputs, and states (see Appendix B.2). All values lie during training in a hypercube between 0 and 1. Therefore, one can check if from different initial conditions $\underline{\hat{x}}(0)$ for different constant levels of the scaled input signal $\tilde{u}(k)$, an equilibrium is reached, a limit cycle is entered, or if the behavior is divergent. In other terms, we check if all relevant points in the state space are in the region of attraction to possibly multiple equilibrium points or limit cycles. This analysis gives an intuition of the model's stability but is surely no mathematical stability proof or guarantee.

As an example, the Bouc-Wen benchmark is used (see Sect. 5.5). The LMSSN model is chosen to have $n_x = 3$ state variables. It is trained and validated with a random phase multisine with $N = 8192$ data points each. Training is done in batch mode with the adaptive moment estimation (ADAM) optimizer (see Appx. A.3), with a base learning rate of $\alpha = 1 \cdot 10^{-3}$. The learning rate is decreased with increasing training epochs by the factor $\frac{1}{\sqrt{i}}$, where i is the number of epochs. The LMSSN has MISO state equations and a MISO output equation, meaning that each state variable and the output are composed of individual LMNs. The obtained LMSSN model has a total number of 68 LMs. Two projections (one of the $\hat{x}_1(k)$–$\hat{x}_2(k)$ plane and another of the $\hat{x}_2(k)$–$\hat{x}_3(k)$ plane) for the initial $N = 128$ steps are shown in Fig. 3.26. The trajectories' beginnings are indicated by the color blue and turning more and more towards yellow for increasing time steps. Two distinct limit cycles can be discerned at input amplitudes of around $u = 0.25$ and $u = 0.6$. The ability of the LMSSN model to produce limit cycles can be understood as an undesirable or desirable property, depending on the point of view. Of course, the models' flexibility to produce those is undesirable for processes where inherently no limit cycles occur. On the other hand, for processes that do have inherent limit cycles (like the Van der Pol oscillator), this shows that the LMSSN is able to identify those as well. An illustrative study of the Van der Pol oscillator is given in Sect. 5.2.

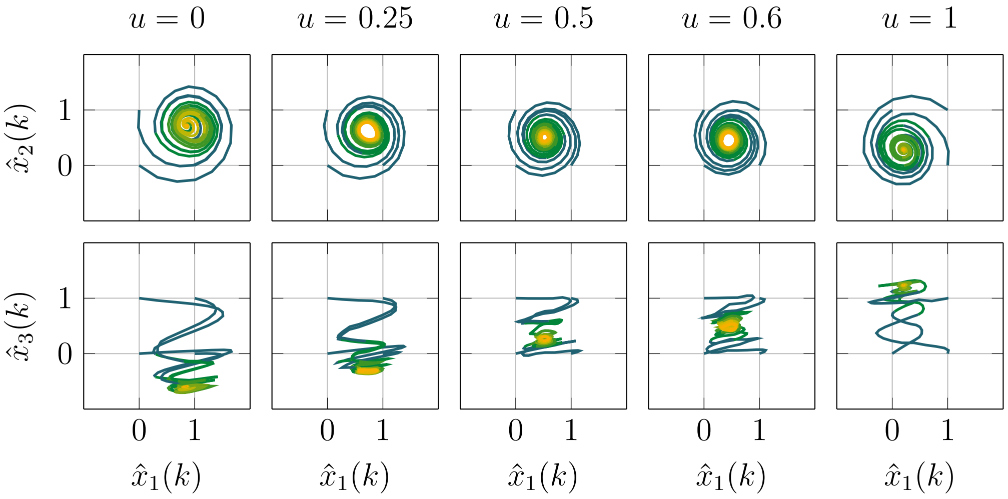


Figure 3.26: Settling behavior of LMSSN with different initial conditions $\underline{\hat{x}}(0)$ (shown are the first 128 time steps). The upper row shows the projection into the $\hat{x}_1(k)$–$\hat{x}_2(k)$ plane, the lower row the projection into the $\hat{x}_1(k)$–$\hat{x}_3(k)$ plane. As initial conditions, the corners of the unit square of the shown projection are chosen with the not shown state variable being equal to 0 ($\underline{\hat{x}}(0) = [0,1]^3$, top row: $\hat{x}_3(0) = 0$, bottom row: $\hat{x}_2(0) = 0$). Each column shows the two projections for different constant levels of the input signal.

3.5 Relation to Other Model Structures and Incorporation of Prior Knowledge

It is beneficial if a proposed model structure can incorporate prior knowledge. This wish comes from the basic rule in estimation to *not estimate what you already know* [130]. In this way, the space of possible model structures is restricted, likely leading to a more accurate and reliable model.

A common way is to distinguish between at least three levels of prior knowledge, which have been color-coded as follows [130]:

- White-box models: The model is fully obtained by physical insight and prior knowledge.
- Gray-box models: There is some physical insight available, but not all parameters are determined by first principles and remain for estimation. Sjöberg et. al. (1995) further divide this category into two sub-classes:
  - Physical modeling: The model is built on physical grounds, with a certain number of parameters to be estimated from data. This could, e.g., be a state space model of given order and structure.
  - Semi-physical modeling: Physical insight is used to suggest certain nonlinear combinations of measured data signals, which can be understood as feature engineering. These new features are then subjected to model structures of black-box character.
- Black-box models: No physical insight is available or used, but the chosen model structure belongs to families that are known to have good flexibility and have been "successful in the past".

Restricting certain properties of the LMSSN imposes more structure on the model, which can be motivated by additionally available (physical) knowledge, making the LMSSN then fall into the semi-physical instead of the black-box modeling category. Those restrictions can be used to show how the LMSSN simplifies to existing approaches (Sect. 3.5.1) and how knowledge can be incorporated to draw connections to block-oriented modeling structures (Sect. 3.5.2).

3.5.1 Differences to Existing Approaches

In the following, we want to look at the differences of the LMSSN to other state space model structures, which employ LMNs or which the LMSSN can degrade to like PWA state space models.

Other Local Model State Space Models There are mainly two contributions, which deal with local model approaches in state space [139, 152]. The main differences to both works are summarized in the following three points.

Structure Identification In both works there is no effort put into structure identification but only on parameter estimation for a given structure. It is assumed that the number, position, size, and shape of validity functions is known a priori. For black-box identification, the RBFs are therefore placed most of the time uniformly in the input space (becoming infeasible for higher-dimensional problems) or an a-priori optimized structure is assumed to be known. In contrast, this thesis focuses on both structure and parameter identification.

Individual Nonlinear Structure of State Variables Both works follow the idea of blending multiple affine state space models (see Sect. 2.1.2.3). This means that the same validity functions are used for state and output equation and that all matrices are blended as a whole. The LMSSN is much more flexible as state variables can be treated individually.

Scheduling with the States Even though it is stated in both works that the validity functions can depend on the state variables and dynamic input, all shown studies only deal with partitioning along the input $u(k)$ and process output $y(k)$ axis. In this way, the authors intend to guarantee stability of gradient calculations if the model itself is stable. This comes at the cost, though, of losing flexibility in the state equation. The LMSSN takes the risk of unstable gradients into account for the gain of nonlinearities within state variables.

Piecewise Affine Models The PWA state space model is introduced in Sect. 2.1.2.3. One major difference to the LMSSN is that all parameters switch simultaneously when the operating point changes from one model to another. In this respect, PWA and multiple affine state space models are equivalent. In contrast, for an LMSSN with a bank of MISO LMNs modeling the state equations, a change of the operating point may lead to altering parameters in an arbitrary number of state equations and not necessarily in all of them. This is the case, as each state is modeled with a unique LMN with possibly different input spaces $\underline{z}$ and individual input space partitionings.

Additionally, a PWA state space model switches *hard* from one model to another. There is no smooth transition between the models, nor is it ensured that two neighboring submodels form a continuous or even differentiable surface. The LMSSN always possesses a smooth transition between the LMs and hence a continuous and differentiable surface.

In one special case, the LMSSN does resemble a PWA state space model. If there is only one set of validity functions that partition the input spaces of all state and output equations equally and if the smooth transition of validity functions becomes abrupt. This can either be achieved by infinitely small standard deviations of the RBFs (for NRBF validity functions) or infinitely large values of the scaling parameter κ for the hierarchical sigmoid functions, see (2.52).

3.5.2 Block-oriented Structures

The LMSSN is designed to be a black-box identification procedure with a minimal number of hyperparameters that *have to* be specified a priori (besides the desired model order, nothing else needs to be specified beforehand). Nevertheless, if prior knowledge is available, it is convenient and advantageous if incorporating this prior knowledge into the identification procedure is possible.

The LMSSN offers the possibility to restrict certain properties of the model structure so that it resembles (or turns into an only slightly more expressive structure than) the block-oriented structures from Fig. 2.14. Restrictions can be imposed in two ways. First, only certain equations can be set to be modeled by LMNs. In turn, all other equations are modeled linearly. Second, if an equation is chosen to be modeled by an LMN, it can be restricted in the input dimensions in which it is allowed to perform splits. It is thus achieved to separate the input space in two sets of input dimensions that influence the model output linearly or nonlinearly (see also Fig. 3.15). Additionally, input dimensions to each equation may be neglected (setting the corresponding parameters to 0 in the LMs) regardless of the equation being a linear function or an LMN. The computational effort is thus drastically reduced and the likelihood to converge to the true process is enhanced.

Wiener Process A Wiener process can be described by

$$\begin{aligned} r(k) &= G(q)u(k) \\ y(k) &= f(r(k))\,, \end{aligned} \tag{3.41}$$

where $G(q)$ is a linear time-invariant (LTI) block of order m and $f(\cdot)$ an arbitrary static function. This process can be modeled in principle by any nonlinear state space model by choosing the order of the state space model the same as the dynamical order of the Wiener process $n_x = m$. To restrict the LMSSN's flexibility to resemble the Wiener process (see Fig. 3.27), the state equation can be chosen as a linear function, while the output equation is modeled as a MISO LMN which only takes the state vector as input as

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \underline{A}\,\underline{\hat{x}}(k) + \underline{b}\,u(k) \\ \hat{y}(k) &= g(\underline{\hat{x}}(k)) \qquad \text{with} \qquad g(\underline{\hat{x}}(k)) = \sum_{l=1}^{n_o} \Big(p_l + \underline{c}_l^T\,\underline{\hat{x}}(k)\Big) \cdot \Phi_l^{[o]}(k)\,. \end{aligned} \tag{3.42}$$

In principle, an even more restrictive model could be chosen, which also imposes structure on $\underline{A}$ and $\underline{b}$ by using canonical representations. If, for example, the observability canonical form is enforced for the state equation, it is only necessary to model a nonlinear output equation with a single state variable as input. Unfortunately, the restriction to assume only one state variable to be a nonlinear input to the output equation is not feasible. Due to the non-uniqueness of the state space formulation, it is unclear in which state variable the nonlinearity will be identified. This challenge could be overcome by the aforementioned transformation into, e.g., the observability canonical form. However, the LMSSN is already transformed after each split to scale the state trajectory to the unit hypercube for the estimation of the validity functions, making it thus not possible to perform an additional transformation without violating this trajectory scaling condition. Therefore, one can only restrict the LMSSN

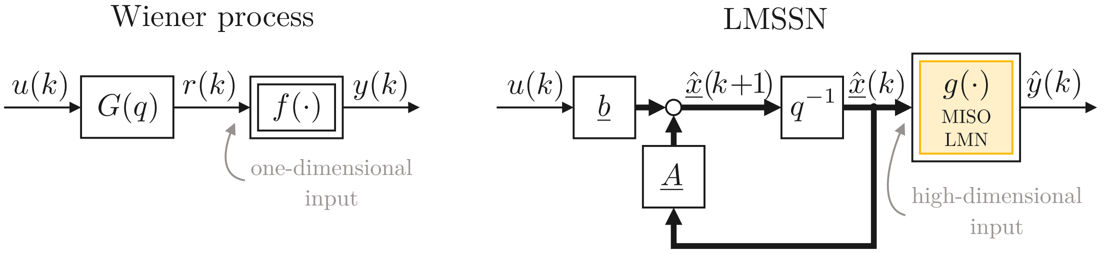


Figure 3.27: Wiener process (left) and LMSSN resembling Wiener process (right)

to splits in all state dimensions and not more specifically to splits in a single state variable.

This makes the restricted LMSSN come close to the Wiener structure but is still more expressive than it. Note that the static nonlinearity in the Wiener process has a one-dimensional input space ($f : \mathbb{R} \to \mathbb{R}$), while the static nonlinearity of the LMSSN has a high-dimensional input space ($g : \mathbb{R}^{n_x} \to \mathbb{R}$). This relaxation to consider nonlinearities in all state variables instead of a single one will also be present in the other block-oriented structures.

Hammerstein Process A Hammerstein process can be described by

$$\begin{aligned} s(k) &= f(u(k)) \\ y(k) &= G(q)s(k)\,, \end{aligned} \tag{3.43}$$

with $G(q)$ of order m and $f(\cdot)$ being an arbitrary static function. Equations (3.43) can be described with an LMSSN of order n_x with a state equation which is nonlinear in the process input $u(k)$ and linear in the state variables and with a linear output equation (see Fig. 3.28) as

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \underline{A}\,\underline{\hat{x}}(k) + h(u(k)) \qquad \text{with} \qquad h(u(k)) = \sum_{j=1}^{n_m} (o_j + b_j u(k)) \cdot \Phi_j^{[s]} \\ \hat{y}(k) &= \underline{c}^T \underline{\hat{x}}(k)\,. \end{aligned} \tag{3.44}$$

As for the Wiener process, it is not clear which state variable (or which combination of state variables) might capture the nonlinear behavior. Therefore, the process input must be an input to all state equations (represented in Fig. 3.28 by the product $h\left(u(k)\right) \cdot \underline{1}$).

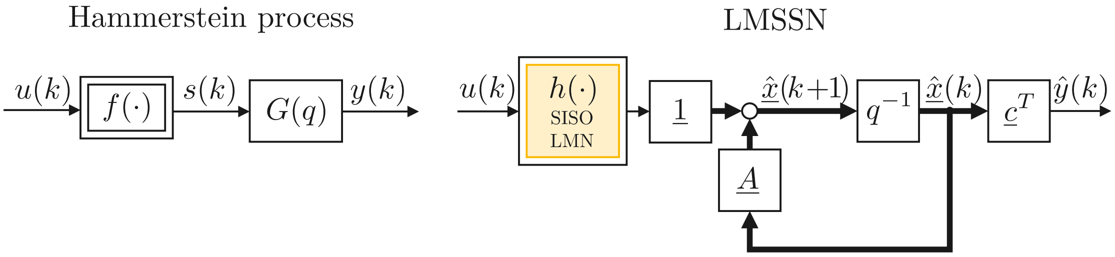


Figure 3.28: Hammerstein process (left) and LMSSN resembling Hammerstein process (right)

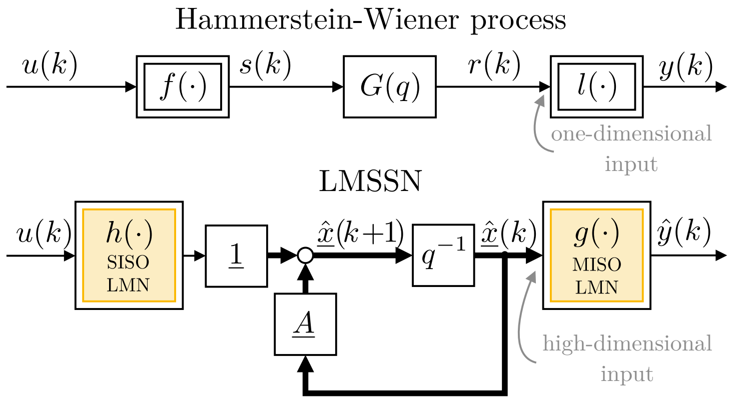


Figure 3.29: Hammerstein-Wiener process (top) and LMSSN resembling Hammerstein-Wiener process (bottom)

Hammerstein-Wiener Process The Hammerstein-Wiener process can be described by

$$\begin{aligned} s(k) &= f(u(k)) \\ r(k) &= G(q)s(k) \\ y(k) &= l(r(k)) , \end{aligned} \tag{3.45}$$

where, again, $G(q)$ is of order m and $f(\cdot)$ and $l(\cdot)$ are arbitrary (static) functions. It can be modeled as a straightforward extension of the Hammerstein process with a nonlinear output equation in the state variables as

$$\begin{aligned} \underline{\hat{x}}(k+1) = \underline{A}\,\underline{\hat{x}}(k) + h(u(k)) \qquad \text{with} \qquad h(u(k)) = \sum_{j=1}^{n_m} (o_j + b_j u(k)) \cdot \Phi_j^{[s]} \\ \hat{y}(k) = g(\underline{\hat{x}}(k)) \qquad \text{with} \qquad g(\underline{\hat{x}}(k)) = \sum_{l=1}^{n_o} \left(p_l + \underline{c}_l^T\,\underline{\hat{x}}(k)\right) \cdot \Phi_l^{[o]}(k) \,. \end{aligned} \tag{3.46}$$

The block diagram of the Hammerstein-Wiener process and the corresponding LMSSN are shown in Fig. 3.29.

Wiener-Hammerstein Process A Wiener-Hammerstein process can be described by

$$\begin{aligned} r(k) &= G(q)u(k) \\ s(k) &= f(r(k)) \\ y(k) &= H(q)s(k) , \end{aligned} \tag{3.47}$$

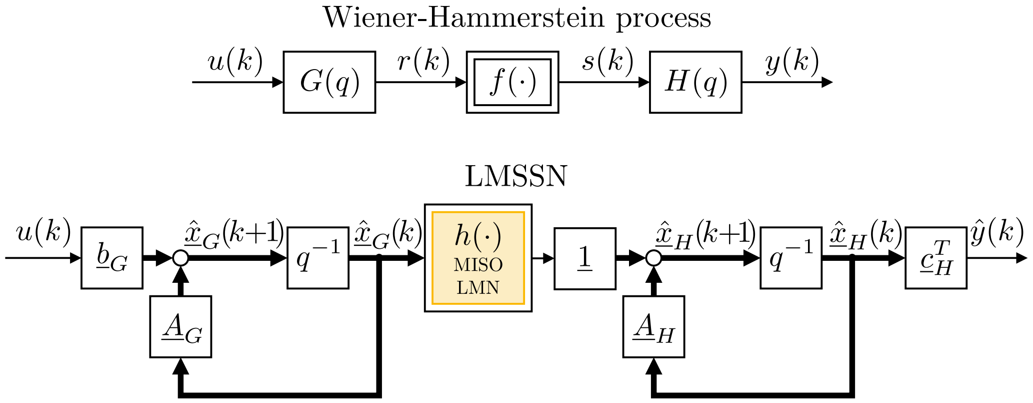


Figure 3.30: Wiener-Hammerstein process (top) and LMSSN resembling Wiener-Hammerstein process (bottom)

where the LTI block $G(q)$ and $H(q)$ are assumed to both be of dynamical order m and $f(\cdot)$ is an arbitrary (static) function. To model such a process, an LMSSN with $n_x = 2m$ state equations is needed. The LMSSN can be set up as

$$\begin{aligned}
\begin{bmatrix}\hat{\underline{x}}_G(k+1)\\ \hat{\underline{x}}_H(k+1)\end{bmatrix} &= \begin{bmatrix}\underline{A}_G & \underline{0}\\ \underline{0} & \underline{A}_H\end{bmatrix}\begin{bmatrix}\hat{\underline{x}}_G(k)\\ \hat{\underline{x}}_H(k)\end{bmatrix} + \begin{bmatrix}\underline{0}\\ \underline{1}\cdot h(\hat{\underline{x}}_G(k))\end{bmatrix} + \begin{bmatrix}\underline{b}_G\\ \underline{0}\end{bmatrix} u(k)\\
y(k) &= \begin{bmatrix}\underline{0} & \underline{c}_H^T\end{bmatrix}\begin{bmatrix}\hat{\underline{x}}_G(k)\\ \hat{\underline{x}}_H(k)\end{bmatrix}\,, \qquad (3.48)\\
\text{with} \qquad h(\hat{\underline{x}}_G(k)) &= \sum_{j=1}^{n_m}\left(\underline{o}_G + \underline{A}_{HG,j}\hat{\underline{x}}_G(k)\right)\cdot \Phi_j^{[s]}\,,
\end{aligned}$$

where each submatrix and vector has appropriate dimensions to either model the dynamic behavior of $G(q)$ or $H(q)$, denoted by the subscripts $(\cdot)_G$ and $(\cdot)_H$, respectively (see Fig. 3.30). Only the lower block of the state equation models nonlinear behavior regarding $\hat{\underline{x}}_G(k)$. The scalar function $h(\cdot)$ is learned just as was done for the Hammerstein process. A lot of sparsity is achieved in the otherwise fully parametrized LMSSN.

Summary of Block-oriented Structures The four different block-oriented processes and how the LMSSN can resemble these is summarized in Tab. 3.1. Here, the focus is on the different input dimension $\tilde{\underline{u}}(k) = [\hat{\underline{x}}(k)\, u(k)]$ to the state and output equation and whether they influence the output of the equation in a linear or nonlinear way.

Table 3.2 summarizes how the number of parameters is reduced from a general

Table 3.1: Overview of how the LMSSN can resemble block-oriented processes. The inputs to the state and output equation are shown and how those exert linear and nonlinear influence on the equation output.

LMSSN Structure	Inputs to state equation $\underline{\hat{x}}(k+1) = \underline{h}(\underline{\hat{\boldsymbol{x}}}(\boldsymbol{k}), \boldsymbol{u}(\boldsymbol{k}))$		Inputs to output equation $\hat{y}(k) = g(\underline{\hat{\boldsymbol{x}}}(\boldsymbol{k}), \boldsymbol{u}(\boldsymbol{k}))$	
	State variables $\underline{\hat{\boldsymbol{x}}}(\boldsymbol{k})$	Process input $\boldsymbol{u}(\boldsymbol{k})$	State variables $\underline{\hat{\boldsymbol{x}}}(\boldsymbol{k})$	Process input $\boldsymbol{u}(\boldsymbol{k})$
LMSSN	Nonlinear	Nonlinear	Nonlinear	Nonlinear
Wiener	Linear	Linear	Nonlinear	-
Hammerstein	Linear	Nonlinear	Linear	-
Hammerstein-Wiener	Linear	Nonlinear	Nonlinear	-
Wiener-Hammerstein	Linear/ Nonlinear	Linear	Linear	-

Table 3.2: Number of parameters of an LMSSN that resembles block-oriented structures. For simplicity, an LMSSN of order n_x with a MIMO state equation and a MISO output equation is assumed. Additionally, it is assumed that each LMN has the same number n_m of LMs.

LMSSN Structure	Number of parameters	Typical LMSSN with $n_x = 3$ and $n_m = 10$
LMSSN order n_x	$(n_x+1)(n_x+2)n_m$	200
Wiener	$(n_x+1)(n_x+n_m)$	52
Hammerstein	$(n_x+1)n_x+2n_m$	32
Hammerstein-Wiener	$(n_x+1)n_m+2n_m+n_x^2$	69
Wiener-Hammerstein	$(n_x+1)n_m+n_x+n_x^2/2$	≈ 48

LMSSN to block-oriented structures. For simplicity, we assume that we deal with an LMSSN of order n_x with a MIMO state equation and a MISO output equation. Additionally, it is assumed that each LMN has the same number n_m of LMs. It can be seen from the table that the general nonlinear identification task can be significantly simplified since, in all cases, the number of parameters is drastically reduced in contrast to the general LMSSN. The number of parameters are explicitly calculated for a typical LMSSN with three state variables and ten LMs in the right column. Note that in the Wiener-Hammerstein case, the LMSSN order is chosen twice the order of the LTI blocks $n_x = 2m$, which always leads to an integer for the number of parameters. How the domain and range of state and output equation are reduced for the LMSSN resembling block-oriented structures can be found in Appx. B.4.

3.6 Numerical and Computational Aspects

In the following, some numerical and computational aspects of the LMSSN are discussed.

3.6.1 Numerical Issues with NRBF Validity Functions

Since RBFs are exponentially decaying, their normalization becomes increasingly harder with increasing distance to their center coordinates. Eventually, when the numerical values of the RBFs fall below the smallest value that can be represented by the computer in the chosen numeric data type, a division-by-zero occurs. To circumvent this problem, when the NRBFs are constructed, the smallest value that can be represented by the chosen numeric data type, ϵ, is added to the denominator as

$$\Phi_j(\underline{u}, \underline{\mu}_j, \underline{\sigma}_j) = \frac{\Psi_j(\underline{u}, \underline{\mu}_j, \underline{\sigma}_j)}{\sum_{s=1}^{n_m} \Psi_s(\underline{u}, \underline{\mu}_s, \underline{\sigma}_s) + \epsilon} . \tag{3.49}$$

For double-precision floating-point numbers, ϵ is round about $\epsilon \approx 2.2 \cdot 10^{-308}$. For demonstration purposes, consider a neural network with two inputs. Now, one additional RBF is added at a time in the u_2-dimension (see Fig. 3.31). This is done in such a way that the region $u_1, u_2 \in [0, 1] \times [0, 1]$ is governed by an increasing number of RBFs (with the aim to transform them eventually to NRBFs). Constant ϵ is set to $\epsilon \approx 2.2204 \cdot 10^{-16}$, which is the relative double-precision floating-point accuracy at the numeric value 1. The colored dots represent the center coordinates of the RBFs. The corresponding colored circular and elliptical lines represent the position in the input space (contour lines) where $\Psi_i = \epsilon$. This means if only a single RBF existed, the NRBF has the function value of 0.5 at this point (since $\Psi_1 = \epsilon$). The NRBF is inside the circle or the ellipse ≈ 1 and outside ≈ 0 (if no other RBFs exist). For an increasing number of RBFs, the standard deviation in the u_2-dimension is decreased, making the validity region of each RBF gradually smaller. This means that with an increasing number of RBFs in one dimension, the region without numerical issues becomes smaller. Let us fix, for example, the coordinate in dimension u_1 to 0.5. In the case with only one RBF, the model has no numerical problems for

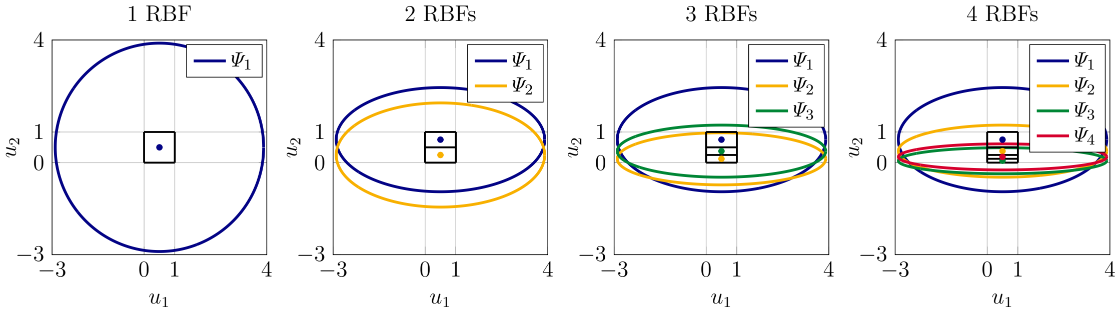


Figure 3.31: RBFs for different numbers of splits in one LMN.

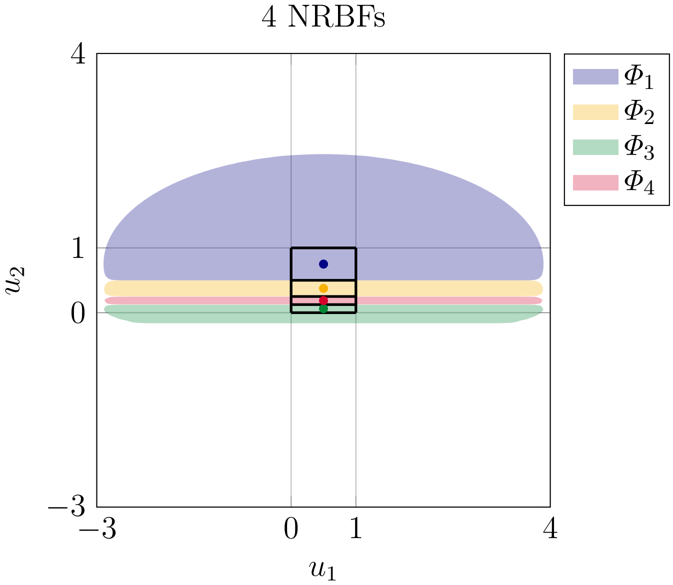


Figure 3.32: Reactivation with NRBFs in extrapolation. The blue model reactivates for values $u_2 \leq -0.2$, where the green model should usually be active.

roughly $-2.9 \leq u_2 \leq 3.9$. The model with four RBFs has no numerical problems for $-0.9 \leq u_2 \leq 2.4$. Therefore, it is likely for a model with many neurons in one dimension that numerical problems (validity functions which are supposed to be active become 0) occur closer to the extrapolation boundaries than for models with only fewer neurons.

Another problem that can occur with NRBF networks is reactivation [128]. It can occur when the underlying RBFs have different standard deviations. The example with four RBFs from Fig. 3.31 is taken and its NRBFs are shown in Fig. 3.32.

It can be seen that the blue model, which is supposed to be active only for high values of u_2, also reactivates for values $u_2 \leq -0.2$. So the model extrapolates for small u_2 not as expected with the green model but with the blue model.

Case Study: Numerical Issues of LMSSN with NRBF Validity Functions Let us consider (3.49) and analyze the influence of this NRBF validity function construction mechanism on dynamical models. Let us take, for example, the state equation of a first-order time-discrete process

$$\hat{x}(k+1) = -a\hat{x}(k) + bu(k)\,, \tag{3.50}$$

where $p = -a$ is the pole and the gain is calculated by $K = \frac{b}{1+a}$. Now, (3.50) is transformed to an LMN with one LM as

$$\hat{x}(k+1) = [-a\hat{x}(k) + bu(k)]\,\Phi(\hat{x}, u, \underline{\mu}, \underline{\sigma})\,. \tag{3.51}$$

The center coordinates in the $\hat{x}$-dimension and u-dimension of the RBF are $\underline{\mu} = [0.5, 0.5]^T$, the standard deviation is set to $\underline{\sigma} = [0.4, 0.4]^T$. This corresponds to the setup of the LMSSN with NRBF validity functions when it is initialized with the BLA.

Now we set the pole of the process to $p = -a = 0.9$ and the gain to $K = 10$ (leading to $b = 1$) and compare the step responses of the process and model described in (3.50) and (3.51). The step responses to process and model are shown in the upper plot of Fig. 3.33. It can be seen that the true process converges slowly to its gain, while the LMSSN periodically turns 0 after six time steps. This behavior originates from the addition of ϵ in the denominator of (3.49). Once the RBF values approach ϵ, the NRBF validity function is no longer 1 but decreases rapidly to 0.

This is demonstrated in the lower plot of Fig. 3.33. Here, the output $\hat{x}(k+1)$ is shown with respect to $\hat{x}(k)$, while the input $u(k)$ is the step function like for the upper plot. For $\hat{x}(k) \geq 3.5$, the validity function decreases from 1 to 0, producing the oscillatory effect. It is therefore crucial that ϵ is chosen as small as possible to extend the valid region of the LMs as far as possible. Depending on the process under test, this problem is sometimes of minor importance, while for other processes, it turns out that LMSSNs with NRBF validity functions are hardly identifiable. This problem is fully overcome with hierarchical sigmoid validity functions at the cost of longer computation times because of their complex construction.

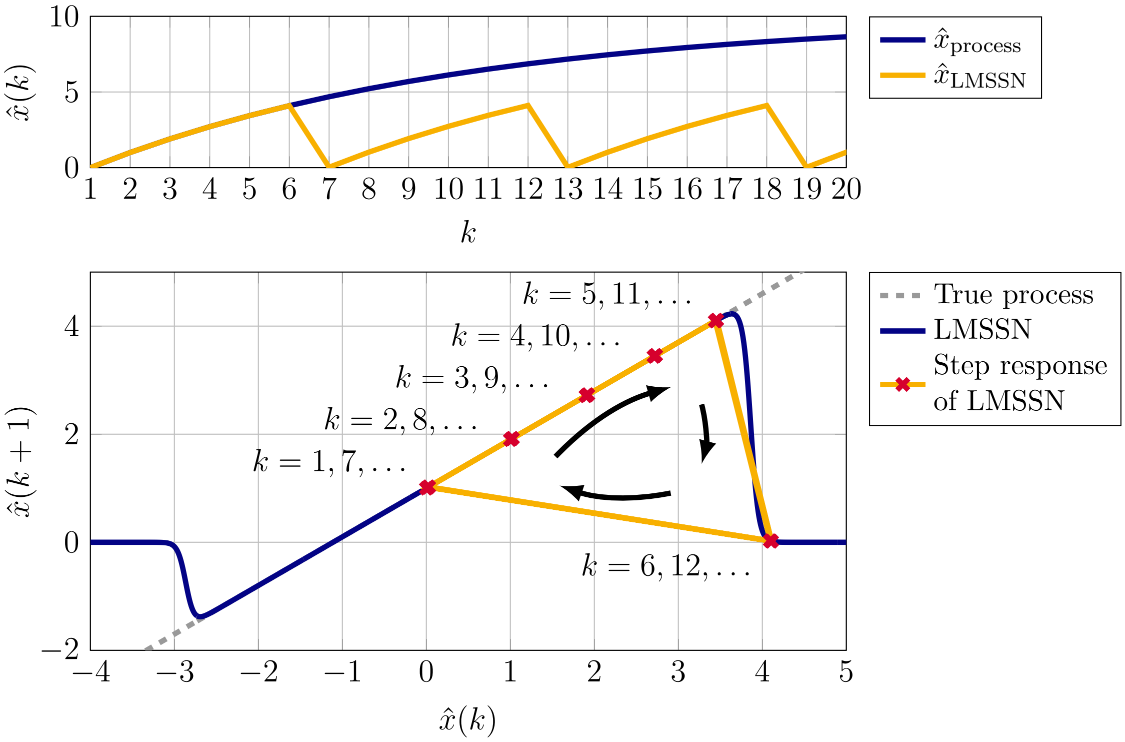


Figure 3.33: Step responses of first-order process and LMSSN model with NRBF validity functions. The upper plot shows the time responses and the lower the state equation. The LMSSN shows oscillatory behavior since the constant ϵ is added during the construction of the NRBFs to prevent division-by-zero.

3.6.2 Computational Aspects

LMSSN Toolbox The user-friendliness of nonlinear system identification algorithms is often underestimated or neglected [122]. Therefore, for the LMSSN algorithm, an object-oriented MATLAB® toolbox is developed, which makes the LMSSN readily available with concise demos and documentation. Another implementation is done in Python using the TensorFlow (v2.6.0) [1] and Keras [20] library.

Gradient Computation The LMSSN MATLAB® toolbox has different options that are used for optimization. Either the Levenberg-Marquardt algorithm (see Sect. 2.3.2) or a Quasi-Newton method (see Appx. A.3) is employed for optimization.

The gradients are either supplied analytically (derivations of the gradients see Appx. B.3) or estimated by finite differences. Due to the recurrent nature and complexity of the gradients, the calculation speed is oftentimes not faster when the gradients are supplied analytically.

The Python version of the LMSSN makes use of the TensorFlow framework, which allows for automatic differentiation and the use of advances from the machine learning community. A comparison of different optimizers for the LMSSN and their performance can be found in [126].

Computation Time The LMSSN is computationally quite expensive. Since multiple LMNs may exist (say there are n_{LMN} LMNs within the same model), where each LMN can be split (in the axis-orthogonal case) in all its inner input dimensions $n_x + n_p$, up to $n_{\text{LMN}} \cdot (n_x + n_p)$ splits and nonlinear optimizations are performed per iteration. The splits can be conveniently and efficiently implemented on parallel architecture, nevertheless leading to much longer computation times than other nonlinear system identification algorithms due to the growing tree structure.

4 Deep Recurrent Neural Networks

Parts of the chapter have been published in [123].

Learning of dynamical models is not a discipline solely confined to system identification. Especially dynamics realizations of state space character are established in the machine learning community, where dynamics are realized by recurrent neural networks (RNNs) [44]. The RNN is quite similar to the nonlinear state space model but not exactly the same. The techniques used in the machine learning field are usually suited for a broader range of problems than in the system identification field. RNNs, especially long short-term memory (LSTM) networks [55], are used for automatic translation or even for music generation. The task in system identification is different, as the system is excited by a possibly multivariate sampled signal and also the output is a signal with the same constant sampling frequency. Recently, so-called transformer architectures have become the state-of-the-art architecture in natural language processing, which was commonly the domain of RNNs [138]. They use attention mechanisms (which can be seen under certain circumstances as a kind of nonlinear finite impulse response (NFIR) structure) and have proven to be very successful on machine translation tasks [27, 12]. Transformers eliminate the time-recurrent structure of the RNN, which speeds up model training drastically and makes enormous numbers of parameters feasible (the well-known language model GPT-3 has 175 billion parameters), as no backpropagation-through-time (BPTT) is necessary. On the downside, the notion of “past and future” is lost (transformer structures are usually non-causal), making transformers in their current form unsuitable for simulation of time-series data, which is the scope of this work.

Therefore, we will focus on RNN structures. Several works have considered the usage of RNNs for the modeling of dynamic behavior. The first works assessing the learning of dynamic systems use time-delayed output values and are similar to nonlinear autoregressive with exogenous input (NARX) models [87]. Also, relatively early internal states for neural networks have been considered [28]. These approaches have also been compared experimentally on relatively simple system identification

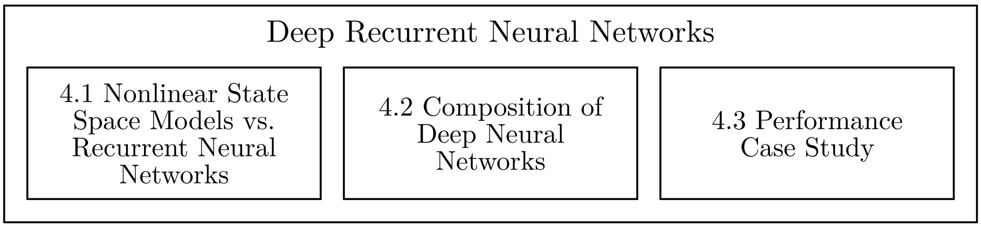

Deep Recurrent Neural Networks
4.1 Nonlinear State Space Models vs. Recurrent Neural Networks
4.2 Composition of Deep Neural Networks
4.3 Performance Case Study

Figure 4.1: Structure of this chapter

tasks [56]. Recently, a deep neural network has been trained for a soft robotics actor [95].

Compared to relatively early approaches to applying neural networks for system identification, we take a different viewpoint. Usually, only a moderate amount of data has been used for the identification of neural networks, say $1000 - 10\,000$ data points. One significant advantage of RNNs trained by stochastic gradient descent (SGD) is that a very large number of data samples, say $10^6 - 10^7$, can easily be used due to the utilization of mini batches (see Sect. 2.3.4). The different possibilities for constructing deep RNNs for nonlinear system identification are analyzed systematically. An overview of this chapter is given in Fig. 4.1.

4.1 Nonlinear State Space Models Versus Recurrent Neural Networks

Let us consider the single-input single-output (SISO) nonlinear state space model from (3.1) and compare it to an RNN structure which is typically described by

$$\begin{aligned} \underline{\hat{x}}_r(k) &= \underline{h}_r(\underline{\hat{x}}_r(k-1), u(k)) \\ \hat{y}(k) &= g_r(\underline{\hat{x}}_r(k)) \,. \end{aligned} \tag{4.1}$$

The RNN block diagram in contrast to the state space model is depicted in Fig. 4.2. Both structures are quite alike and it is possible to transfer the state space structure to an RNN representation and vice versa. If a state space structure is given, the

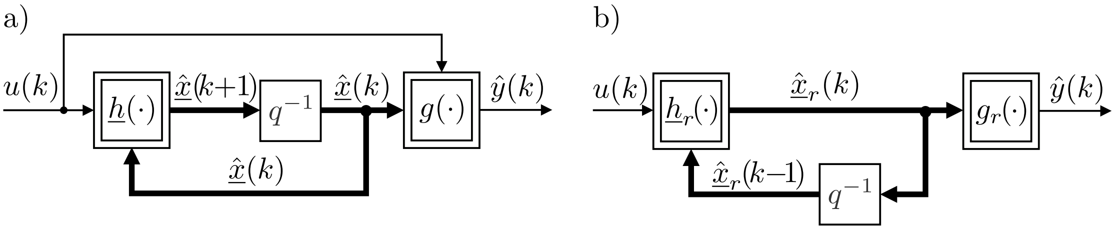


Figure 4.2: Similarity between a) the nonlinear state space and b) the RNN.

corresponding equations describing the RNN can be found by

$$\begin{aligned}\underbrace{\begin{bmatrix}\hat{\underline{x}}(k)\\ \hat{x}_u(k)\end{bmatrix}}_{\hat{\underline{x}}_r(k)} &= \underbrace{\begin{bmatrix}\underline{h}([\hat{\underline{x}}(k-1)\ \hat{x}_u(k-1)]^T)\\ u(k)\end{bmatrix}}_{\underline{h}_r(\hat{\underline{x}}_r(k-1),u(k))} \\ \hat{y}(k) &= g(\underbrace{[\hat{\underline{x}}(k)\ \hat{x}_u(k)]^T}_{\hat{\underline{x}}_r(k)})\,.\end{aligned} \tag{4.2}$$

In the RNN state equation, for calculating the current state $\hat{\underline{x}}_r(k)$, the one time step delayed input $u(k-1)$ is only implicitly included in $\hat{\underline{x}}_r(k-1)$. To incorporate this time delay explicitly, $u(k)$ is added to the state equation as additional state variable $\hat{x}_u(k)$, which is an argument to the original state equation $\underline{h}(\cdot)$ in its time-delayed version.

If the RNN structure is given, then the equations of the state space model can be found by

$$\begin{aligned}\underbrace{\tilde{\underline{x}}_r(k+1)}_{\hat{\underline{x}}(k+1)} &= \underbrace{\underline{h}_r(\tilde{\underline{x}}_r(k), u(k))}_{\underline{h}(\hat{\underline{x}}(k),u(k))} \\ \hat{y}(k) &= \underbrace{g_r(\underline{h}_r(\tilde{\underline{x}}_r(k), u(k)))}_{g(\hat{\underline{x}}(k),u(k))}\,,\end{aligned} \tag{4.3}$$

with the new state $\tilde{\underline{x}}_r(k)$, which is a one time step delayed version of the state of the RNN so that $\tilde{\underline{x}}_r(k) = \hat{\underline{x}}_r(k-1)$.

4.2 Composition of Deep Recurrent Neural Networks

This section explains the RNN structure and its components in more detail. Note that for the recurrent state $\hat{\underline{x}}_r(k)$, the subscript $(\cdot)_r$ is dropped. Normal weight arrows indicate vector signal flows for the rest of the chapter as only recurrent models and vectorial quantities are dealt with.

4.2.1 Simple Recurrent Layer

A simple RNN layer (with similarities to the state equation of an affine state space model) is shown in Fig. 4.3. A hidden state in layer i, $\underline{\hat{x}}_i(k)$, is calculated by a linear combination of its one time step delayed version $\underline{\hat{x}}_i(k-1)$ and the input to the layer at the current time step $\underline{\hat{x}}_{i-1}(k)$, which is then passed through a nonlinearity. Note that q^{-1} indicates the time-shift operator and that the input to the first layer is simply $\underline{\hat{x}}_{i-1}(k) = u(k)$. Hereby, $\underline{\hat{x}}_i(k-1)$ is weighted with the parameter matrix $\underline{A}_i \in \mathbb{R}^{n_{x_i} \times n_{x_i}}$, $\underline{\hat{x}}_{i-1}(k)$ is weighted with $\underline{B}_i \in \mathbb{R}^{n_{x_i} \times n_{x_{i-1}}}$. Additionally, an offset or bias vector $\underline{o}_i \in \mathbb{R}^{n_{x_i} \times 1}$ is added before the linear combination is passed through a nonlinearity, which is for RNNs oftentimes a hyperbolic tangent (see also Appx. A.1), denoted by $\sigma(\cdot)$. The hyperbolic tangent is chosen for its smoothness and its continuous differentiability. The output of the i-th layer of a deep RNN at time step k is therefore

$$\underline{\hat{x}}_i(k) = \sigma(\underline{o}_i + \underline{A}_i\, \underline{\hat{x}}_i(k-1) + \underline{B}_i\, \underline{\hat{x}}_{i-1}(k))\,. \tag{4.4}$$

This notation is chosen to illustrate the similarities to a state space model. Likewise, illustrations of RNNs such as in [45], which underline the computer science perspective on the neurons, will not be shown here. Instead of unfolding the network over time, it is more intuitive in controls to use the time-shift operator q^{-1}.

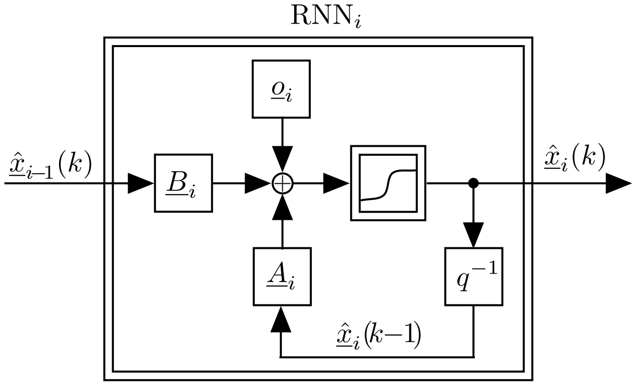

RNN$_i$
$\underline{o}_i$
$\underline{\hat{x}}_{i-1}(k)$
$\underline{B}_i$
$\underline{\hat{x}}_i(k)$
$\underline{A}_i$
q^{-1}
$\underline{\hat{x}}_i(k-1)$

Figure 4.3: Illustration of simple recurrent layer

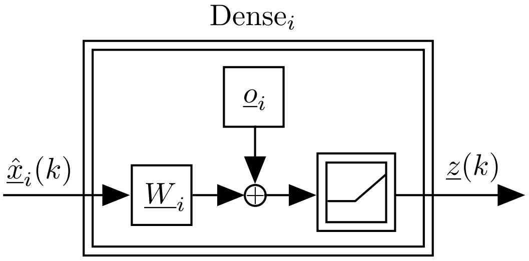


Figure 4.4: Illustration of a dense layer

4.2.2 Fully Connected Layer (Dense Layer)

A fully connected layer (or dense layer) is a simple feed-forward layer, see Fig. 4.4. It maps the input vector (for example $\hat{\underline{x}}_i(k)$) weighted with the matrix $\underline{W}_i \in \mathbb{R}^{n_{x_i} \times n_z}$ and an offset $\underline{o}_i \in \mathbb{R}^{n_z \times 1}$ after passing a rectified linear unit (ReLU) activation to the output of the layer $\underline{z}(k)$

$$\underline{z}(k) = \mathrm{ReLU}(\underline{W}_i \hat{\underline{x}}_i(k) + \underline{o}_i)\,. \tag{4.5}$$

The ReLU is defined as

$$\mathrm{ReLU}(\underline{x}) = \max(0, \underline{x})\,, \tag{4.6}$$

where the max($\cdot$) operator is applied element-wise. The ReLU is a popular choice as it avoids the vanishing gradient problem and is computationally less expensive than the hyperbolic tangent or sigmoid. A drawback of the ReLU is that it is not differentiable at the origin. The coverage of the unique advantages and disadvantages of many other existing activation functions is beyond the scope of this thesis.

4.2.3 Connecting the Layers

An almost infinite number of possibilities exists for the composition of deep RNNs. To conquer this challenge, a set of network variations is chosen from one default setup, depicted in Fig. 4.5. The input $u(k)$ passes n series-connected RNN layers, each with n_{x_i} neurons (or in terms of the state space models with n_{x_i} state variables). After that, one standard fully connected layer (n_z neurons and a ReLU activation, see Fig. 4.4) maps the last state $\hat{\underline{x}}_n(k)$ to $\underline{z}(k)$. To form an output that can also have negative values, $\underline{z}(k)$ is weighted with the matrix $\underline{W}_o \in \mathbb{R}^{1 \times n_z}$ (here a vector) to finally obtain the scalar model output $\hat{y}(k)$.

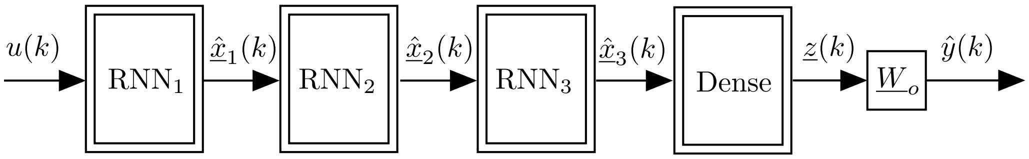


Figure 4.5: Illustration of deep recurrent neural network

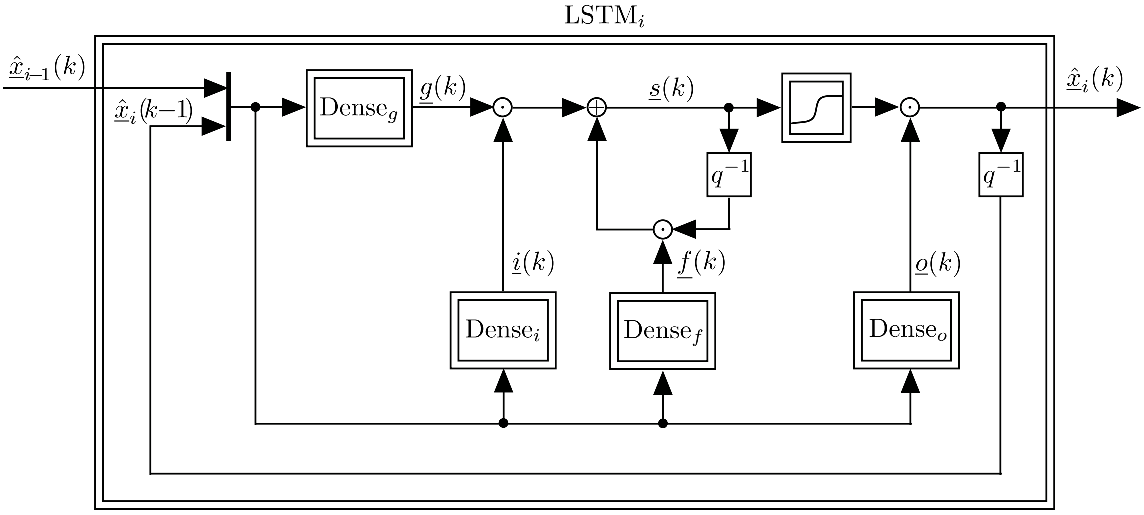


Figure 4.6: Illustration of an LSTM layer

4.2.4 Complex Recurrent Layer Networks

Instead of connecting many simple layers to achieve a sufficiently flexible model, one could also use fewer but more complex recurrent layers.

Long short-term memory Probably the most commonly known complex RNN layer is the LSTM layer [55], depicted in Fig. 4.6. The key idea of LSTMs is to build an internal state $\underline{s}(k)$ into each neuron. This internal state acts as a memory cell that can retain information over a period of time, dependent on the forget gate activation $\underline{f}(k)$. The forget gate activation acts as the pole of a first-order dynamic system, with stable dynamics (due to the sigmoid activation in the forget gate, the pole is always in the range (0 1)). The forget gate activation thus determines the dynamics of the internal cell and how much information can be retained for how long [89].

The input to the layer $\hat{\underline{x}}_{i-1}(k)$ and the one time step delayed output of the current layer $\hat{\underline{x}}_i(k-1)$ are fed through four different dense layers, with the same number of neurons n_{x_i}. Usually, for Dense$_g$ a hyperbolic tangent is chosen as an activation function and for the other three dense layers, a sigmoid is a common choice [66]. The signal $\underline{g}(k)$ passes first the input gate, indicated by the Hadamard product of $\underline{g}(k) \odot \underline{i}(k)$. Understanding this operation as a gate is intuitive, as $\underline{i}(k)$ lies between 0 and 1 due to the sigmoid activation of the previous time step. This gate, therefore, determines how much of the input signal is let through.

As previously mentioned, the forget gate activation $\underline{f}(k)$ influences the dynamics of the so-called internal state $\underline{s}(k)$ and can be thought of as the pole of a first-order discrete-time system for the dynamics of $\underline{s}(k)$. It determines how much of the past information is memorized or forgotten. Then, the signal is passed through a nonlinearity, usually a hyperbolic tangent, and then multiplied with the output gate activation $\underline{o}(k)$, leading to the layer output $\underline{\hat{x}}_i(k)$. The LSTM illustration in Fig. 4.6 bridges the gap between the different understandings of dynamic modeling in the computer science and controls disciplines. Unlike most other publications, the here presented LSTM shows the model from the controls perspective.

GRU An alternative popular complex recurrent layer is the gated recurrent unit (GRU) [19]. In general, it has fewer parameters than the LSTM as it only possesses two instead of three gates. A GRU layer from the system identification perspective is shown in Fig. 4.7. The output of the layer $\underline{\hat{x}}_i(k)$ is a tradeoff between a *proposed state* $\underline{\tilde{x}}_i(k)$ and the state of the previous time step $\underline{\hat{x}}_i(k-1)$. This tradeoff is controlled by the update gate activation $\underline{z}(k)$ as

$$\underline{\hat{x}}_i(k) = \underline{z}(k) \odot \underline{\tilde{x}}_i(k) + (\underline{1} - \underline{z}(k)) \odot \underline{\hat{x}}_i(k-1)\,. \tag{4.7}$$

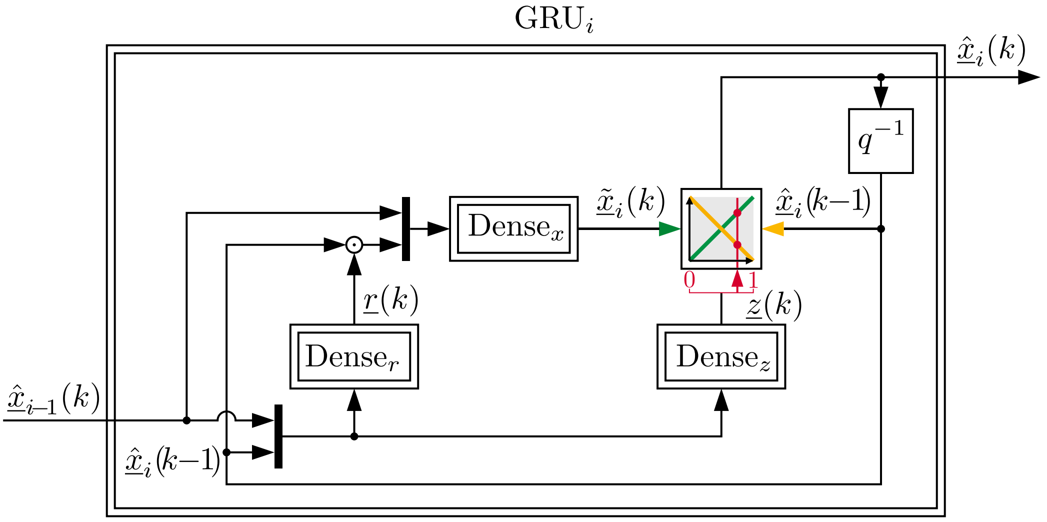

GRU_i
$\underline{\hat{x}}_i(k)$
q^{-1}
$\underline{\tilde{x}}_i(k)$
$\underline{\hat{x}}_i(k{-}1)$
Dense_x
$\underline{r}(k)$
0
1
$\underline{z}(k)$
Dense_r
Dense_z
$\underline{\hat{x}}_{i-1}(k)$
$\underline{\hat{x}}_i(k{-}1)$

Figure 4.7: Illustration of a GRU layer

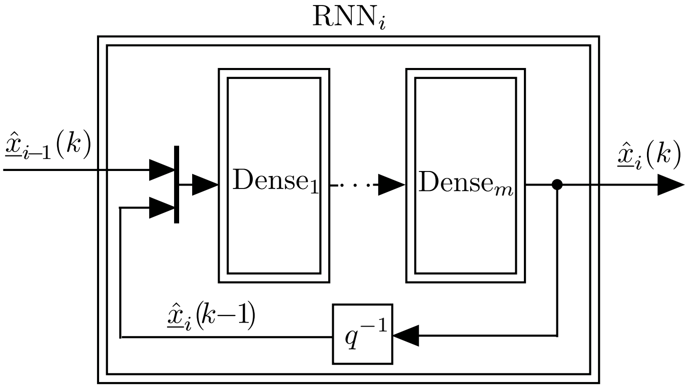

RNN$_i$
$\hat{\underline{x}}_{i-1}(k)$
Dense$_1$
Dense$_m$
$\hat{\underline{x}}_i(k)$
$\hat{\underline{x}}_i(k-1)$
q^{-1}

Figure 4.8: Illustration of complex recurrent layer

The proposed state $\tilde{\underline{x}}_i(k)$ combines the input to the layer $\hat{\underline{x}}_{i-1}(k)$ and the previous state of the current layer $\hat{\underline{x}}_i(k-1)$ that are passed through a dense layer. The reset gate activation $\underline{r}(k)$ controls the amount the previous state contributes to the proposed state.

For the reset and update gate, commonly sigmoid functions are used (to have the interpretability as gates as their outputs are in the range 0 to 1), while for the proposed state layer Dense$_x$, a hyperbolic tangent is used.

Other Complex Recurrent Layers Other, more complex, RNN layers can be modeled, such as in Fig. 4.8. In this case, the recurrent layer does not only comprise a simple affine function followed by a nonlinearity but is replaced by multiple dense layers. The output after all dense layers is then fed back. It constitutes a more sophisticated layer type than shown in Fig. 4.3, but this extension is straightforward compared to the LSTM or GRU layer.

4.3 Case Study: Bouc-Wen Benchmark

Different deep RNN structures will be studied on the Bouc-Wen benchmark. The full description of the benchmark can be found in Sect. 5.5. This benchmark allows generating own training and validation data. Therefore, a random phase multisine input signal with $N = 500\,000$ data points with excited frequencies between $5 - 150$ Hz was created and applied to the simulated benchmark system. The obtained dataset was then split into sequences of size $N_s = 1000$ with 50 % overlap leading to a total of 999 sequences. The first 70 % of the sequences was used as training data and the

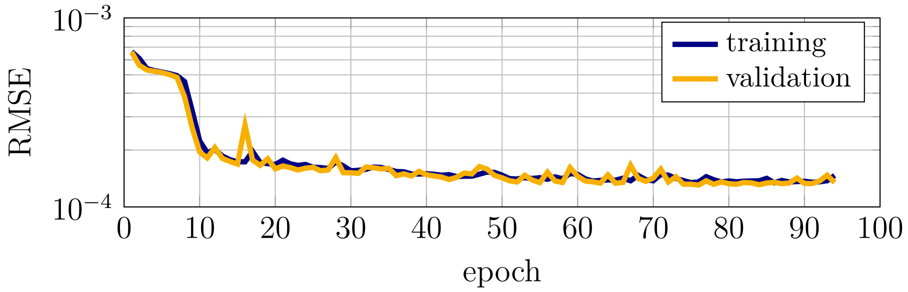


Figure 4.9: Convergence of default deep RNN on training and validation data

remaining 30 % as validation data. The sampling frequency is $f_s = 750\,\text{Hz}$. The root mean squared (rms) amplitude of the input is 50 N (force). The benchmark provides two fixed test datasets, one with a random phase multisine input ($N = 8192$) and the other one with a sine-sweep input ($N = 153\,000$).

The models are compared by their root mean squared error (RMSE) on test data, which is calculated by

$$\text{RMSE} = \sqrt{\frac{1}{N_t}\sum_{k=1}^{N_t}(y_t(k) - \hat{y}_t(k))^2}\,. \tag{4.8}$$

Here, $y_t(k)$ is the process output on test data, $\hat{y}_t(k)$ is the model output on test data, and N_t is the number of test samples. All results shown here were obtained using TensorFlow (v2.6.0) [1] with the Keras library [20] using the following setup. The default training routine uses the adaptive moment estimation (ADAM) optimizer (see Appx. A.3) [59] with a learning rate of $1 \cdot 10^{-3}$. All networks are trained for 100 epochs if the early stopping criterion (RMSE on validation data did not improve over the last 15 epochs) is not met before. The loss function is chosen to be the RMSE on training data.

The default network architecture (Fig. 4.5) has $n = 3$ hidden RNN layers (with $n_{x_i} = 30$ neurons each) followed by a fully connected layer with $n_z = 100$ neurons and the output weight matrix, mapping the n_z outputs of the dense layer to the final model output. This default setup leads to a total of 7820 trainable parameters. An exemplary convergence plot is shown in Fig. 4.9. Here, after 94 epochs, the algorithm stopped. As the ADAM algorithm is a stochastic optimizer, it can be seen that the loss function shows slight fluctuations on training data and larger fluctuations on validation data over the advancing epochs.

The RMSE on the multisine test signal is $\text{RMSE} = 7.6 \cdot 10^{-5}$. The process is captured

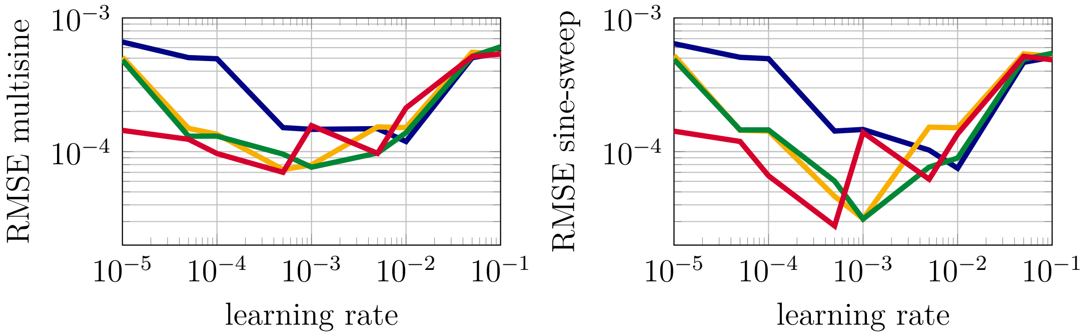


Figure 4.10: RMSEs for different learning rates and a different number of neurons (▬) $n_x = 5$, (▬) $n_x = 30$, (▬) $n_x = 50$,(▬) $n_x = 100$ per layer on mulitsine and sine-sweep test datasets.

well with the default deep RNN. The training of this default network converged for all trials that were run. On the sine-sweep dataset, the RMSE is $4.1 \cdot 10^{-5}$, which is also a good result when compared to other identification algorithms (see Tab. 5.4 at the end of Sect. 5.5).

4.3.1 Variation of Learning Rate

For this study, the learning rate was varied between $1 \cdot 10^{-5}$ and $1 \cdot 10^{-1}$. In Fig. 4.10 on the left, the results for networks with three simple recurrent layers (default setup), each with 5, 30, 50, or 100 neurons is shown. This leads to architectures with 845 up to 60 600 parameters. It can be seen that learning rates around $1 \cdot 10^{-3}$ lead to the best results on the multisine test signal. It can also be seen that model performance increases with the number of neurons per layer. For 30 neurons and more, though, the performance does not increase significantly anymore. The same can be said for the results on the sine-sweep test dataset (see Fig. 4.10 on the right). Here, also the learning rate of $1 \cdot 10^{-3}$ yields the best results and the differences in performance of 30 up to 100 neurons per layer are also negligible.

4.3.2 Number of Neurons and Layers

In the default setup, for all three layers, the number of neurons is varied from $n_{x_i} = 2$ up to $n_{x_i} = 50$. This leads to parameter numbers for the overall network ranging from 428 to 17 900. The results can be seen in Fig. 4.11 (left side) on the sine-sweep test dataset. With an increasing number of neurons, better model qualities can be

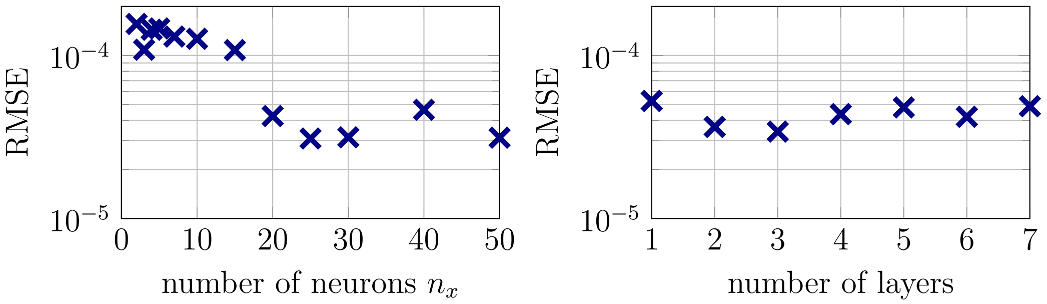


Figure 4.11: RMSE for different numbers of neurons on the sine-sweep test dataset

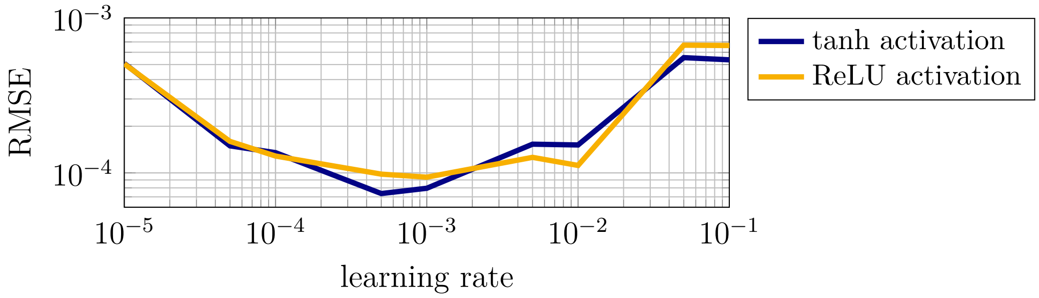


Figure 4.12: RMSE for different activation functions on multisine test dataset.

achieved. Especially $n_{x_i} = 25$ and $n_{x_i} = 30$ seem to establish a parsimonious model with satisfactory model complexity.

Another study was carried out with a variation in the number of recurrent layers that are series-connected. All layers have the default number of $n_{x_i} = 30$ neurons. The results are shown in Fig. 4.11 on the right. Even though parameter numbers range from 4161 (one recurrent layer) to 15 141 (seven recurrent layers), performance on the sine-sweep test signal stays in the same range. The best performance is obtained for a network with three recurrent layers. For more than seven layers, the models were not able to converge anymore. This might be due to the vanishing gradient problem [54].

4.3.3 Comparison of Different Activation Functions

Common choices for activation functions in recurrent layers are the hyperbolic tangent or the ReLU [66]. Both will be compared when used in each of the three recurrent layers. The results are summarized in Fig. 4.12. It can be seen that even though the activation functions are quite different, the results for different learning

Table 4.1: Comparison of different number of series-connected LSTM layers

	RMSE multisine $\cdot 10^{-5}$	RMSE sine-sweep $\cdot 10^{-5}$
default RNN	7.64	4.12
1 LSTM layer	6.32	3.16
2 LSTM layers	6.10	**2.78**
3 LSTM layers	**5.98**	2.80
1 GRU layer	6.57	3.55
2 GRU layers	6.30	5.03
3 GRU layers	6.11	3.86

rates are very similar. Overall, the hyperbolic tangent activation performs slightly better than the ReLU, probably due to its smoothness.

4.3.4 Complex Recurrent Layers

Next, instead of simple RNN layers, complex RNN layers (Fig. 4.8), LSTM layers (Fig. 4.6), and GRU layers (Fig. 4.7) are used. For the complex RNN layer, instead of three series-connected simple recurrent layers, one complex layer, which comprises three dense layers, was used. Within each dense layer inside the complex RNN layer, the number of neurons was varied between 2 and 50. None of the studied model architectures yields a satisfactory model quality as the average RMSE on test data is at least three times greater compared to multiple simple RNN layers. For neuron numbers below 25 per dense layer, the models did not even converge.

The LSTM study was carried out with one, two, and three LSTM layers, each with $n_{x_i} = 30$ neurons. The number of parameters is 7040, 14 360, and 21 680, respectively. The GRU study was also done for one to three layers with 6081, 11 571, and 17 061 parameters, respectively.

The results on the two test datasets are summarized in Tab. 4.1. All three model architectures perform well on the Bouc-Wen benchmark. The networks with two and three LSTM layers perform slightly better than one LSTM layer.

4.3.5 Discussion

A comprehensive comparison of published results on the Bouc-Wen benchmark can be found in Tab. 5.4 at the end of Sect. 5.5. Even though the deep RNN architectures chosen here were not tailored to the overall best performance on the benchmark problem, they lead to acceptable results without tedious tuning. It can be seen that it is possible to achieve state-of-the-art performance by simply increasing the amount of data for training and the depth of the network, which is also true for many other applications like image recognition or language translation. Deep RNNs thus offer promising results also for system identification when large datasets are available.

5 Studies and Applications

This chapter covers studies and applications of the local model state space network (LMSSN) on various processes. The shown studies highlight some characteristics of LMSSNs, while the usefulness and applicability of LMSSN are demonstrated on benchmark problems as well as real-world problems. An overview of this chapter is given in Fig. 5.1.

Error Figures For comparison, usually, the root mean squared error (RMSE) or normalized root mean squared error (NRMSE) is used. The RMSE is calculated by

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(y(k) - \hat{y}(k))^2}\,, \tag{5.1}$$

where N is the number of samples, $y(k)$ is the process output, and $\hat{y}(k)$ is the model output. The NRMSE is calculated as

$$\mathrm{NRMSE} = \sqrt{\frac{\sum_{k=1}^{N}(y(k) - \hat{y}(k))^2}{\sum_{k=1}^{N}(y(k) - \bar{y})^2}}\,, \tag{5.2}$$

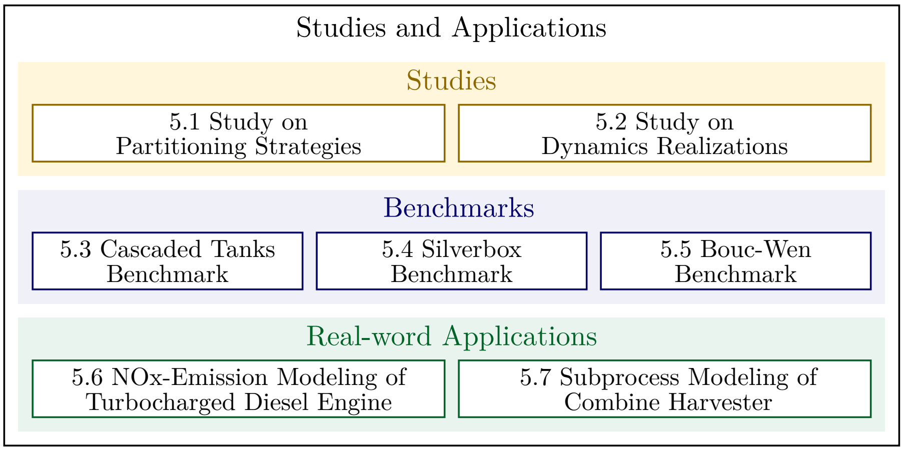

Studies and Applications
Studies
5.1 Study on Partitioning Strategies
5.2 Study on Dynamics Realizations
Benchmarks
5.3 Cascaded Tanks Benchmark
5.4 Silverbox Benchmark
5.5 Bouc-Wen Benchmark
Real-word Applications
5.6 NOx-Emission Modeling of Turbocharged Diesel Engine
5.7 Subprocess Modeling of Combine Harvester

Figure 5.1: Structure of this chapter

where $\bar{y}$ is the mean of the process output. This expression is equivalent to the RMSE divided by the standard deviation of the process output

$$\mathrm{NRMSE} = \frac{\mathrm{RMSE}}{\mathrm{std}(y)}\,. \tag{5.3}$$

The NRMSE can thus be interpreted as the ratio of the standard deviation of the error signal (if the error signal is mean-free) and the standard deviation of the output signal. Alternatively, the NRMSE can be interpreted as the achieved improvement by the model (NRMSE < 1) compared to the baseline model $\hat{y}(k) = \bar{y}$ (which has an NRMSE $= 1$). Note that the error figures can be calculated for different datasets, e.g., on training, validation, or test data.

Default Training The LMSSN is trained with the Levenberg-Marquardt algorithm, see Sect. 2.3.2. Per split, usually 100 Levenberg-Marquardt iterations are carried out. Some implementation details can be found in Appx. A.4. Alternatively, a Quasi-Newton optimizer is used (see Appx. A.3). All shown results are computed with the MATLAB® implementation of LMSSN either on the high-performance OMNI-cluster of the University of Siegen or on a notebook with a 2,3 GHz Quad-Core Intel Core i5 and 16 GB RAM.

Default Termination Criterion The default termination criterion for LMSSN is to terminate if the model performance on validation data does not increase sufficiently anymore. The algorithm terminates if

$$s_c - s_o \geq n_d\,, \tag{5.4}$$

where s_c is the current number of total splits, s_o is the number of total splits until the NRMSE on validation data is in the vicinity of the currently lowest NRMSE value on validation data, and n_d is the number of allowed deteriorations. An example of the criterion is given in Fig. 5.2. All values below a threshold $\mathrm{NRMSE}_{\mathrm{min}} + \mathrm{NRMSE}_{\mathrm{noise}}$ define the vicinity to the lowest NRMSE value. The value $\mathrm{NRMSE}_{\mathrm{noise}}$ is set to $\mathrm{NRMSE}_{\mathrm{noise}} = 0.01$ if not mentioned differently. This corresponds to the assumption that the signal-to-noise ratio (SNR) of the validation output signal is 40 dB. If more knowledge about the signals is available, $\mathrm{NRMSE}_{\mathrm{noise}}$ is adjusted accordingly. If no noise is assumed (SNR $= \infty$ dB), $\mathrm{NRMSE}_{\mathrm{noise}}$ is 0 and the two dashed lines fall onto each other.

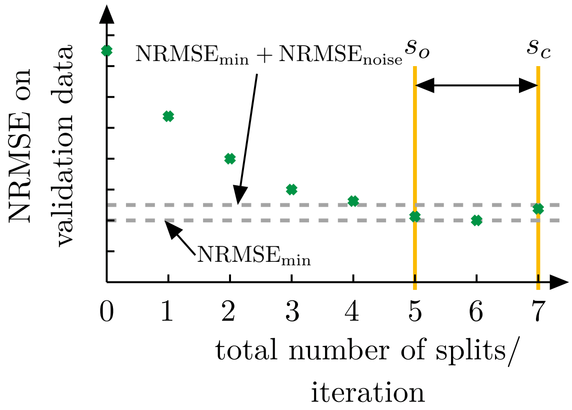


Figure 5.2: Example of default termination criterion. For a user-defined number of deteriorations, e.g., $n_d = 2$, the algorithm terminates after the seventh split. At the fifth split, the NRMSE drops the first time below $\text{NRMSE}_{\text{min}} + \text{NRMSE}_{\text{noise}}$. The algorithm will continue with the next split if n_d is chosen larger.

5.1 Study on Partitioning Strategies

In this study, we investigate different partitioning strategies of the LMSSN. Axis-orthogonal splitting of the local linear model tree (LOLIMOT) and the hierarchical local model tree (HILOMOT) are compared to the axis-oblique partitioning, which is possible for HILOMOT.

To illustrate the differences and for simplicity, a nonlinear first-order process is chosen. The process is described by

$$\begin{aligned} G_1(z) &= \frac{0.6z^{-1}}{1-0.2z^{-1}} \qquad \text{if} \quad y(k) < 1 - u(k)\,, \\ G_2(z) &= \frac{0.4z^{-1}}{1-0.6z^{-1}} \qquad \text{if} \quad y(k) \geq 1 - u(k)\,, \end{aligned} \tag{5.5}$$

with a smooth transition between the two linear time-invariant (LTI) systems realized by weighting them with sigmoid validity functions, see Fig. 5.3. The training, validation, and test dataset are amplitude-modulated pseudo random binary signals (APRBSs) of length $N = 512$, respectively. The training and validation data is corrupted by white Gaussian noise at the output so that those two signals have an SNR of 40 dB.

Three LMSSN models of first order are trained: one with LOLIMOT, one with HILOMOT (axis-orthogonal splits only), and a third model with HILOMOT but

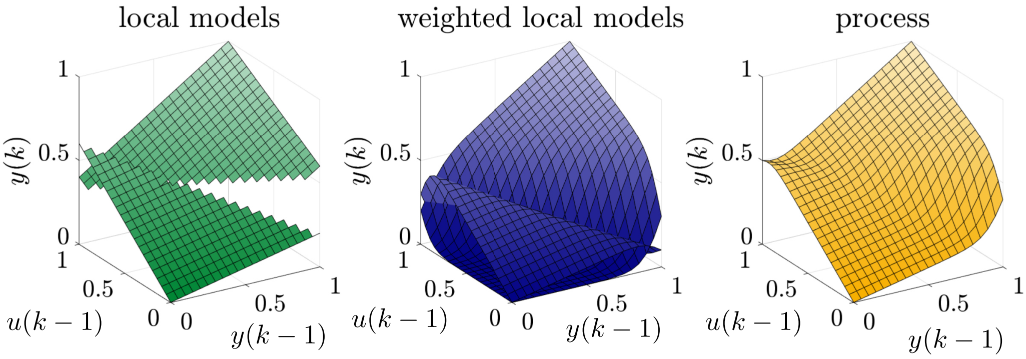


Figure 5.3: The nonlinear first-order example process consists of two LTI systems blended via axis-oblique validity functions in the $(u(k-1), y(k-1))$-space.

with axis-oblique splits possible as well.

The state equation of the LMSSN is modeled by a multiple-input single-output (MISO) local model network (LMN), while the output equation is an affine function. After training, the output equation is transformed to $\hat{y}(k) = \hat{x}(k)$. This is achieved by transforming the output equation parameters to $c^* = 1$ and $p^* = 0$ via the affine state transformation (3.27) with $T = \frac{1}{\hat{c}}$ and $t = -\frac{\hat{p}}{\hat{c}}$. The feedthrough is neglected $(d = 0)$.

The NRMSE on test data, the number of splits, and the mean evaluation time are shown in Fig. 5.4. The axis-orthogonal LOLIMOT and HILOMOT produce roughly the same error on test data. The algorithm stopped in both cases after the third split. The axis-oblique HILOMOT models the artificial process with only one split perfectly. This is expected as the process requires an oblique split in the inner input space $[\hat{x}_1(k)\ u(k)]$.

The LOLIMOT is roughly four times faster than the axis-orthogonal HILOMOT and six times faster than the axis-oblique HILOMOT per split[1]. The main reason why the LOLIMOT model is faster than the HILOMOT model is the construction of the validity functions, which is computationally a lot more involved for hierarchical sigmoid functions than for “unstacked” normalized radial basis functions (NRBFs). For the axis-oblique HILOMOT, the split parameters $\underline{v}_j$ from (2.52) are optimized as well, leading to two additional parameters per split that are optimized. The percentage of additional parameters through axis-oblique partitioning decreases for an overall

[1] using the Levenberg-Marquardt algorithm with analytical gradients; computed on 2.3 GHz Quad-Core Intel Core i5, 16 GB RAM

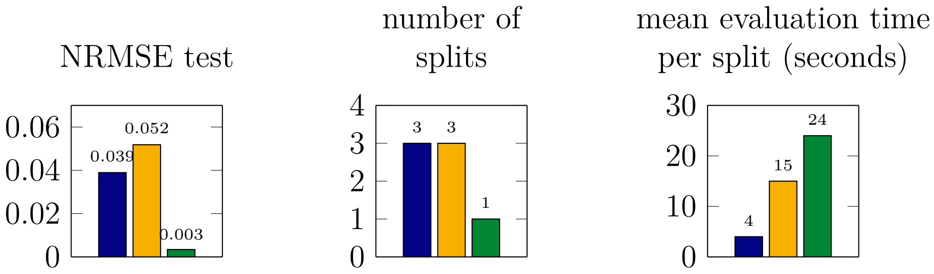


Figure 5.4: Results of the (🟦) LOLIMOT, (🟧) HILOMOT orthogonal, and (🟩) HILOMOT oblique LMSSN model on nonlinear first-order example process.

increasing number of parameters of the LMSSN. Therefore, for models of higher complexity, the computational times for both HILOMOT methods approach each other. This is fundamentally different behavior than for HILOMOT in a nonlinear autoregressive with exogenous input (NARX) or nonlinear finite impulse response (NFIR) model with LMNs as here the local model (LM) parameters are estimated by least squares and only the split parameters have to be optimized nonlinearly. Since all LM and split parameters are estimated nonlinearly in the LMSSN, the nonlinear split optimization does not produce significant additional computational costs. Therefore, it is always recommended to perform nonlinear split optimization when using HILOMOT for greater flexibility at the expense of minor additional computational demand.

The state equation map, as well as the input space partitionings, are shown in Fig. 5.5. The two orthogonal partitioning models are structurally not well suited to correctly model the “oblique” process and look a lot alike concerning their partitionings. Note that the splits are shifted in the $\hat{x}_1$-dimension, which occurs through the rescaling of the state trajectory (see Sect. 3.3.3). The two models do not produce exactly the same results due to their validity function’s slightly different construction mechanisms (see also Fig. 2.16). The oblique partitioning HILOMOT model is able to simulate the process behavior perfectly.

5.2 Study on Dynamics Realizations

In this study, different dynamics realizations shall be compared on the Van der Pol oscillator process. It is described by a second-order ordinary differential equation

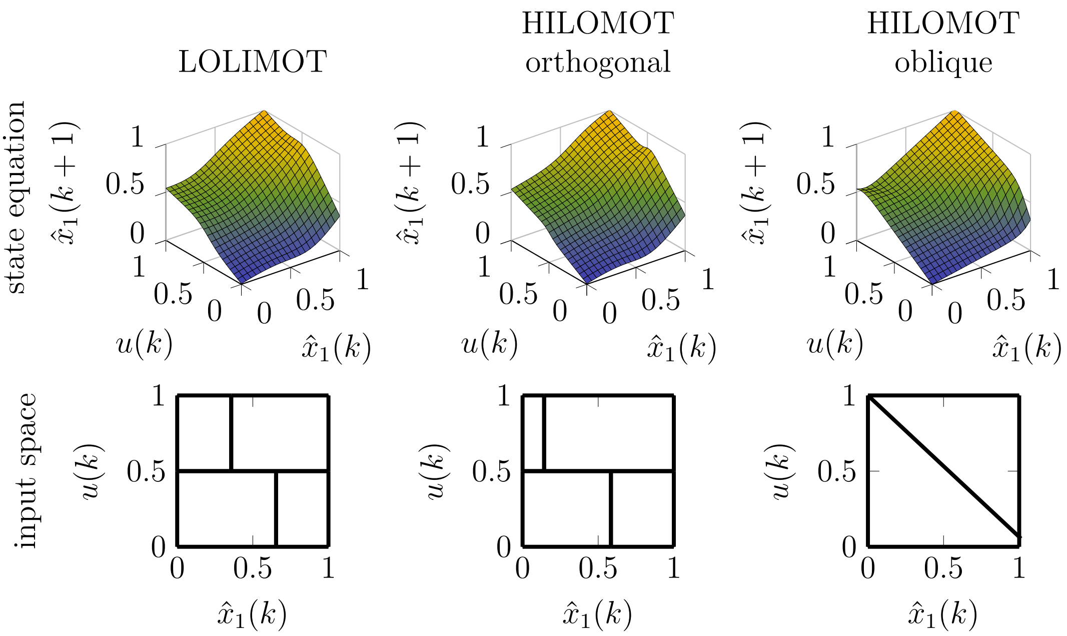


Figure 5.5: Results of three first-order LMSSN models with different construction strategies. The state equation map is shown in the upper row and the input space partitionings are shown in the bottom row for the three models, respectively.

(ODE) with a nonlinear damping term as

$$\ddot{y}(t) + \varepsilon\left(y^2(t) - 1\right)\dot{y}(t) + \omega_0^2 y(t) = u(t)\,. \tag{5.6}$$

Here, $u(t)$ is a forcing term, $y(t)$ is considered the output, and ω_0 is the natural angular frequency [26]. The nonlinear damping term is controlled by ε, called the Van der Pol parameter. The system's damping can be positive or negative, depending on the output level, which leads to limit cycles. Discretizing (5.6) using a first-order forward Euler step yields

$$\begin{aligned} \underline{x}(k+1) &= \begin{bmatrix} 1 & T_s \\ -\omega_0^2 T_s & \varepsilon T_s + 1 \end{bmatrix} \underline{x}(k) + \begin{bmatrix} 0 \\ T_s \end{bmatrix} u(k) + \begin{bmatrix} 0 \\ -\varepsilon T_s \end{bmatrix} x_1^2(k) x_2(k) \\ y(k) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \underline{x}(k)\,, \end{aligned} \tag{5.7}$$

where T_s is the sampling time. The parameters are set to $\varepsilon = 0.03$, $\omega_0 = 2\pi$, and $T_s = 0.01\,\text{sec}$, just as is done in [26]. The Van der Pol oscillator is in itself a polynomial nonlinear state space model (PNLSS) (see also Sect. 2.1.2.3).

The training, validation, and test dataset each contain a random phase multisine with $N = 4096$ data points, all with a root mean squared (rms) value of 50. The training and validation output is corrupted with white Gaussian noise so that the SNR of the signals is 40 dB.

Four different models are trained. The first two models are a NARX and an NFIR model with an LMN as static approximator (trained with the LMNTOOL v1.5.2 [48]). Here, the HILOMOT algorithm with oblique partitioning is used. The order of the NARX model is not chosen equal to the process order since this model performed quite poorly (likely due to missing regressor terms that include cross-products and higher-order terms. It has been found by trial and error that a NARX model with $\underline{s} = [u(k-1)\ u(k-2)\ y(k-1) \ldots y(k-8)]$ and $\underline{z} = [y(k-1) \ldots y(k-4)]$ (compare Sect. 3.3.2 for the explanation of the $\underline{s}$ and $\underline{z}$ input spaces) performs quite well. For the NFIR model, the order and different combinations in $\underline{s}$ and $\underline{z}$ input spaces did not matter performance-wise (all NFIR models perform poorly, explanation see below). The third model is a second-order LMSSN model constructed with the HILOMOT algorithm. The state equation is modeled by MISO LMNs and the output equation is an affine function. For the LMSSN, $\underline{s} = [\hat{x}_1(k)\ \hat{x}_2(k)]$ and $\underline{z} = [\hat{x}_1(k)\ \hat{x}_2(k)]$. The fourth model is a PNLSS model (trained with PNLSS v1.0 [135]). For the state equation, monomial combinations only of the states of second and third order are allowed $\mathcal{O}(\underline{\zeta}) = [2\ 3]$. The output equation is a linear function $\mathcal{O}(\underline{\eta}) = [\ ]$.

The NARX, LMSSN, and PNLSS model results are shown in Fig. 5.6. Since the Van der Pol oscillator process (5.7) is a PNLSS structure, the PNLSS model can correctly and perfectly model the Van der Pol oscillator. The LMSSN has one LM in the first state equation and four LMs in the second state equation. It performs well and is able to identify the process accurately but not perfectly due to the inherent structural differences between the polynomial process and local affine modeling strategy. The NARX model is also able to find a decent description of the process, with some shortcomings, though. It is in principle possible to model the second-order process with a second-order NARX model. However, choosing the model's dynamic order equal to the order of the process led in this study to erroneous models. This is likely due to missing cross-products and higher-order terms in the regressor, as mentioned before. Additionally, the Van der Pol oscillator exhibits at certain frequencies deterministic chaos. Therefore, the exponentially growing errors dominate for models that do not closely resemble the process. Only the increase in (local) model order ($\underline{s}$-space), as well as an increase of splittable dimensions ($\underline{z}$-space), led to acceptable results at the cost of a large number of necessary splits and parameters. This highlights the fact that state space models are much more expressive than input-output models (comparing models of the same order), as clarified in Sect. 2.1.3. All NFIR

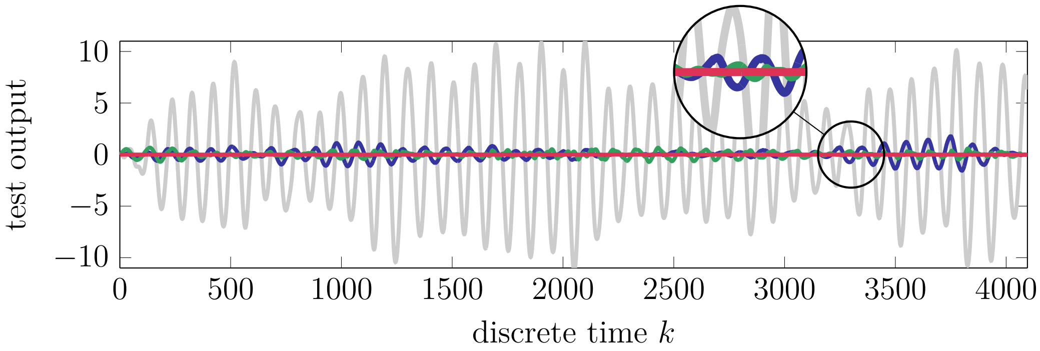


Figure 5.6: Results on Van der Pol oscillator. Shown are the (▬) noise-free test process output of Van der Pol oscillator, (▬) NARX error, (▬) LMSSN error, and (▬) PNLSS error.

Table 5.1: Comparison of NRMSEs of different approaches on Van der Pol oscillator test dataset.

NRMSE	Approach
0.095	NARX LMN ($m = 8$), 33 LMs, 491 parameters
> 1	NFIR LMN (various orders)
0.001	PNLSS, 23 parameters
0.047	LMSSN, 5 LMs, 23 parameters

models failed in modeling the process which is expected as the feedback is lacking which is necessary to model the oscillatory behavior. This is nevertheless a noteworthy drawback as the class of processes that can be modeled with NFIR models is limited. The NRMSE values of the different approaches are summarized in Tab. 5.1.

5.3 Cascaded Tanks Benchmark

The cascaded tanks benchmark is a fluid level control system consisting of two tanks with free outlets fed by a pump. The input signal controls a water pump that transports the water from a reservoir into an upper water tank. The water flows from the upper water tank through a small opening into the lower water tank and finally back into the reservoir. The output of the process is the water level of the lower tank, which is measured with a capacitive water level sensor. This process has a smooth and weak nonlinear behavior. If one of the tanks overflows because of a too high input signal amplitude, the water flows directly into the water reservoir, exhibiting a strong nonlinear behavior [120]. The combination of the weak nonlinear behavior during normal operation and a hard saturation effect for high input signal peaks make this benchmark challenging. Furthermore, only a short data record is available for the benchmark, which poses additional difficulty.

Without the overflow effect, the following state space model can be constructed based on Bernoulli's principle and conservation of mass

$$\begin{aligned} \dot{x}_1(t) &= -k_1\sqrt{x_1(t)} + k_4 u(t) + w_1(t)\,, \\ \dot{x}_2(t) &= k_2\sqrt{x_1(t)} - k_3\sqrt{x_2(t)} + w_2(t)\,, \\ y(t) &= x_2(t) + e(t)\,, \end{aligned} \tag{5.8}$$

where $u(t)$ is the input signal, $x_1(t)$ and $x_2(t)$ are the state variables of the system, $w_1(t)$, $w_2(t)$ and $e(t)$ are noise sources, and k_1, k_2, k_3, and k_4 are constants depending on the properties of the system.

The estimation dataset[2] and test dataset each consist of $N = 1024$ data points. The input signals include excitation in the frequency range from 0 to 0.0144 Hz. The sampling period is $T_s = 4\,\text{sec}$. The measured output signals have an SNR close to 40 dB.

Black-box Modeling A HILOMOT LMSSN is trained, where only axis-orthogonal splits are allowed. Optimization is done with a Quasi-Newton method. The model is of second order ($n_x = 2$) and each state and output equation is modeled by a separate MISO LMN. There is no additional knowledge incorporated. The LMSSN has three LMs in the first state equation, four LMs in the second state equation, and two LMs in the output equation. It achieves an RMSE of 0.52 on the given test dataset (RMSE of the best linear approximation (BLA) is 0.72).

Gray-box Modeling Prior knowledge can be incorporated by assuming block-oriented structures (see Sect. 3.5.2) or by restricting the $\underline{z}$ and $\underline{s}$ input spaces (see Sect. 3.3.2). The latter will be used here. When we neglect the overflow effect, the LMSSN can incorporate the prior knowledge, available from (5.8). The state equation LMNs are only allowed to split in the state dimensions and not in the input dimension $\underline{z}^{[s]}_{1,2} = [\hat{x}_1(k)\ \hat{x}_2(k)]$, while the LM input space contains the full inner input space $\underline{s}^{[s]}_{1,2} = [\hat{x}_1(k)\ \hat{x}_2(k)\ u(k)]$. The output equation is an affine function $\underline{z}^{[o]} = [\,]$,

[2] The estimation data is split into training data (first 70 % of the estimation data) and validation data (last 30 % of the estimation data).

$\underline{s}^{[o]} = [\hat{x}_1(k)\ \hat{x}_2(k)\ u(k)]$. Those restrictions, though, did not yield a significant improvement in modeling accuracy.

Hard Restriction Now the LMSSN is given exactly the flexibility implied by the nonlinear structure of the physical model. For that, the first LMN is only allowed to split in the first state dimension $\underline{z}_1^{[s]} = [\hat{x}_1(k)]$ while the LM input space is $\underline{s}_1^{[s]} = [\hat{x}_1(k)\ u(k)]$. Therefore, the input can only influence the first state equation linearly. The second state equation (second LMN) is allowed to split in both state variable dimensions $\underline{z}_2^{[s]} = [\hat{x}_1(k)\ \hat{x}_2(k)]$ and there is no influence of the input signal $\underline{s}_2^{[s]} = [\hat{x}_1(k)\ \hat{x}_2(k)]$. The output equation is an affine function only with the second state variable as input $\underline{z}^{[o]} = [\,]$, $\underline{s}^{[o]} = [\hat{x}_2(k)]$.

Applying those restrictions leads to two challenges. First, the BLA returns the best possible linear model. Due to the non-uniqueness of the state space representation, it is unclear if the state variables are in the same “order” as the physical model. Second, the BLA is learned with all model matrices fully populated. Now the BLA needs to be transformed so that certain parameters turn zero. To accomplish this, a similarity transformation is not easily possible. Therefore, the unneeded entries are simply set to zero and then a nonlinear optimization is carried out. The assumption is that the initial BLA is a good starting point to converge to the restricted linear model, but it is not guaranteed that this step is always successful.

When restricting the LMSSN equations to resemble (5.8) perfectly, the model performance improved significantly to an RMSE = 0.25. This error value is currently the best result on this benchmark to the author’s knowledge. Figure 5.7 shows the fully restricted LMSSN output in comparison to the BLA and the process output.

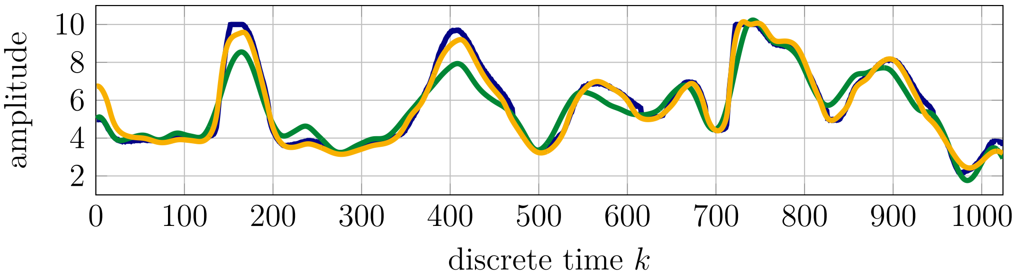


Figure 5.7: Results on the test dataset of the cascaded tanks benchmark. Here, (▬) is the process output, (▬) is the model output of the BLA, and (▬) is the output of the fully restricted LMSSN.

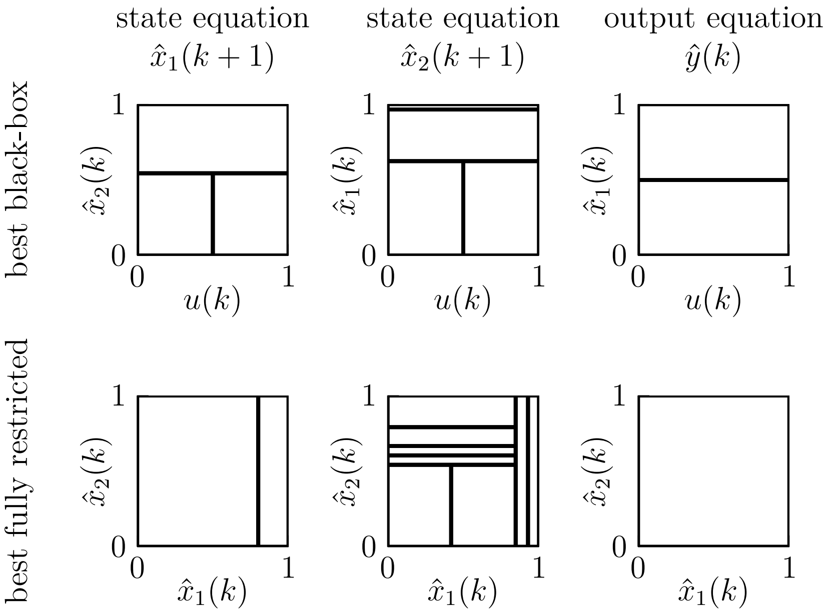


Figure 5.8: Projection of input spaces of best black-box (top row) and best fully restricted LMSSN (bottom row). Here, (—) indicate splits in the input space. Note that all splits are visible and there are no splits in the missing dimension that is not shown in the projection. For the best fully restricted model, the $\underline{z}$ spaces are restricted to $\underline{z}_1^{[s]} = [\hat{x}_1(k)]$, $\underline{z}_2^{[s]} = [\hat{x}_1(k)\ \hat{x}_2(k)]$, and $\underline{z}^{[o]} = [\,]$.

Comparison of Black-box to Fully Restricted Model The black-box LMSSN has 38 parameters. It has three LMs in the first state equation, four LMs in the second state equation, and two LMs in the output equation. The fully restricted LMSSN has a total of 33 parameters. This model has two LMs in the first state equation, eight LMs in the second state equation, and the output equation is affine. Projections of the input spaces of the black-box and fully restricted LMSSN are shown in Fig. 5.8. For the fully restricted LMSSN, it can be seen that the algorithm paid the most attention to the nonlinear modeling of state equation two. The restriction of the $\underline{z}$ and $\underline{s}$ input spaces led to improved modeling quality. In both cases, training was carried out on the OMNI-cluster. The black-box LMSSN training took 185 min, the training of the fully restricted LMSSN only 83 min. The speed-up can be explained by fewer tried splits and fewer optimizable parameters.

Comparison to Other Approaches The overall LMSSN performance on this benchmark is comparable and sometimes better than state-of-the-art nonlinear system identification algorithms. In [146], a very low RMSE of 0.30 is achieved with an extended model obtained with a genetic algorithm. The simulation model was extended

Table 5.2: Comparison of different methods applied to the cascaded tanks benchmark.

RMSE [V]	Approach	Ref.
0.30	Extended model with genetic algorithm	[146]
0.34	Basis function expansion state space model	[105]
0.38	Output error method	[14]
0.45	Flexible nonlinear state space model	[133]
0.45	PNLSS	[105]
0.51	Genetic algorithm	[146]
0.63	NFIR with LMN	[8]
0.67	NOBF with LMN	[8]
0.93	NARX with LMN	[8]
0.71	BLA	
0.52	Best black-box LMSSN	
0.25	Best fully restricted LMSSN	

by a correctional term to better fit this specific process. Without the correctional term, the genetic algorithm achieves an RMSE of 0.51. Other results on this benchmark problem are summarized in Tab. 5.2.

5.4 Silverbox Benchmark

The Silverbox identification problem represents a nonlinear mechanical resonating system, i.e., mass, viscous damping, and nonlinear spring [144]. The electrical circuit which simulates this system is designed to relate the displacement $y(t)$ (the output) to the force $u(t)$ (the input) by the following second-order differential equation

$$m\ddot{y}(t) + d\dot{y}(t) + k_1 y(t) + k_3 y^3(t) = u(t)\,, \tag{5.9}$$

with the mass m, damping factor d, and k_1 and k_3 describing the static but position-dependent stiffness of the nonlinear spring.

The given benchmark input signal contains $N = 131\,072$ data points and consists of two parts (see Fig. 5.9). The first part (blue) is a white Gaussian noise sequence with 40 000 samples filtered by a ninth order discrete-time Butterworth filter with a cut-off frequency of 200 Hz. The amplitude increases linearly over the interval from 0 to its maximum of about 0.1 V. This part will be used for testing since the end of this “arrow tip” tests the models in extrapolation. The second part of the signal

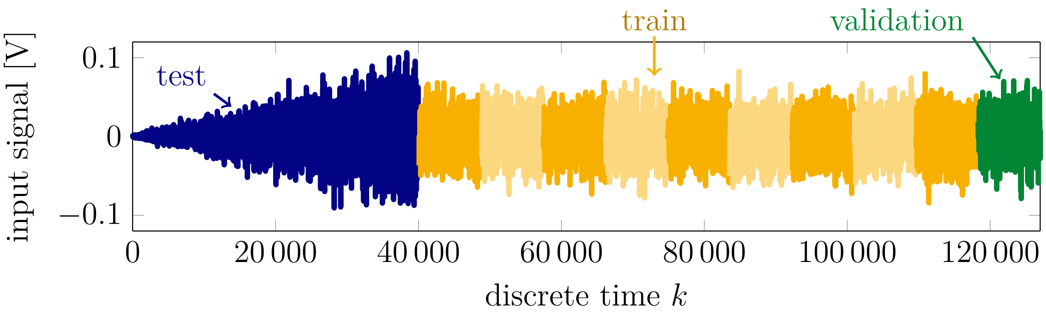


Figure 5.9: Silverbox benchmark input signal

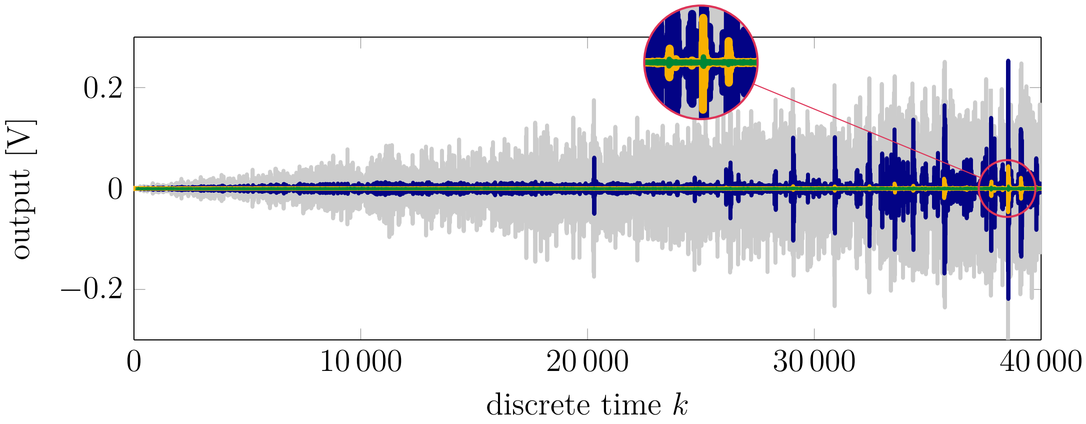


Figure 5.10: Results on the Silverbox test dataset. Shown is (▬) the test process output, (▬) error of the BLA, (▬) error of the MISO/MISO LMSSN, (▬) error of the PNLSS.

(yellow and green) consists of ten successive realizations of a random odd multisine signal. The first nine realizations of the multisine are used for training, while the tenth realization is used as validation dataset.

Four LMSSN models (the four setups from Fig. 3.3 a)-d)) are trained with LOLIMOT (and not HILOMOT) for faster computation on the long training data record. The SNR is assumed to be ∞ dB ($\text{NRMSE}_{\text{noise}} = 0$). The model order is chosen to $n_x = 2$ as the process described in (5.9) can be written as a second-order state space system. The results can be seen in Fig. 5.10. Shown are the test output signal and the errors of the BLA, the MISO/MISO LMSSN, and a PNLSS model with $\mathcal{O}(\underline{\zeta}) = [2\ 3]$ and $\mathcal{O}(\underline{\eta}) = [\ ]$ (trained with PNLSS v1.0 [135]).

The LMSSN achieves an RMSE of 1.26 mV which is a significantly better result than the linear model (RMSE = 13.7 mV). The LMSSN performs well in extrapolation (roughly after 20 000 time steps, while, from here on, the BLA produces high error values. The PNLSS has a very low test RMSE of 0.29 mV. The PNLSS works so well

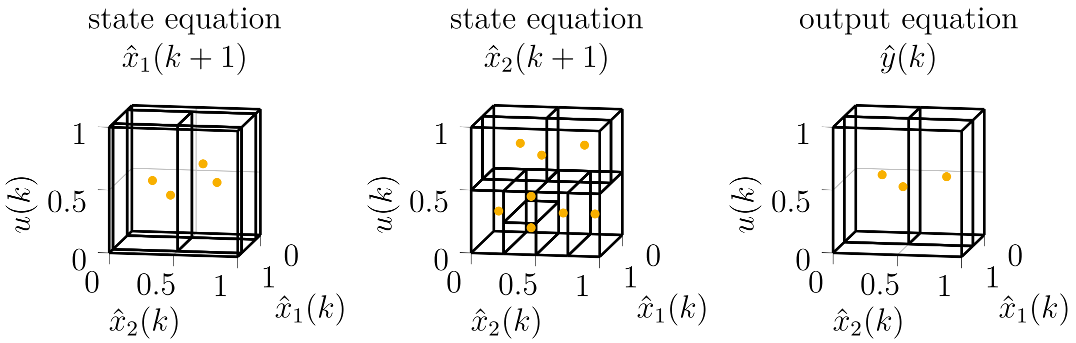


Figure 5.11: Input spaces of the two state equations (with four and eight LMs, respectively) and input space of the output equation (three LMs). The black contours (—) are the borders of the LMs, (●) are the center coordinates of the LMs.

even in extrapolation on this benchmark due to the similarity between the PNLSS model and the internal polynomial structure of the Silverbox [96]. This similarity is a clear benefit on the test dataset for this specific process/model combination. If the polynomial degrees of the PNLSS do not match the process, severe erratic behavior in extrapolation is the consequence (see Sect. 3.4.2).

The partitioning of the input spaces of the two state equations and the output equation of the MISO/MISO LMSSN model are shown in Fig. 5.11. The first state equation has four LMs, the second state equation eight LMs and the output has three LMs.

Table 5.3 summarizes some of the best results obtained on the Silverbox problem by other algorithms. The LMSSN performs quite well, considering that it is a black-box model. Physical block-oriented and specifically to the task tailored models (like the NARX model with custom polynomial regressors), nevertheless perform even better.

5.5 Bouc-Wen Benchmark

Hysteresis is a phenomenon that is often encountered in very diverse engineering and science disciplines. The identification of those systems is challenging, as the output depends on the trajectory of the input. One extensively used model of hysteresis systems is the Bouc-Wen model, which is provided in this benchmark [94].

Table 5.3: Comparison of different methods applied to the Silverbox benchmark.

RMSE [mV]	Approach	Ref.
0.26	PNLSS	[98]
0.30	NARX with custom regressor	[71]
0.32	LSSVM with NARX	[31]
0.35	Poly-LFR	[136]
0.38	Physical block-oriented	[97]
0.96	Physical block-oriented	[53]
1.30	Local Linear State Space	[140]
3.98	LSTM	[5]
4.08	MLP	[5]
4.88	TCN	[5]
7.80	MLP-ANN	[132]
9.10	Extended fuzzy logic	[111]
13.71	BLA	
1.26	MISO / MISO, 12 splits (60)	
1.62	MISO / affine, 7 splits (40)	
1.66	MIMO / MISO, 15 splits (96)	
2.07	MIMO / affine, 10 splits (92)	

The vibrations of a single-degree-of-freedom Bouc-Wen system, i.e., a Bouc-Wen oscillator with a single mass, is governed by Newton's law of dynamics written in the form

$$m\ddot{y}(t) + r(y, \dot{y}) + z(y, \dot{y}) = u(t)\,, \tag{5.10}$$

where m is a mass, $y(t)$ is the displacement, and $u(t)$ is an external force. The total restoring force in the system is composed of a static term $r(y, \dot{y})$ which depends only on the current value of the displacement $y(t)$ and the current value of the velocity $\dot{y}(t)$. The term $r(y, \dot{y})$ is assumed to be linear as

$$r(y, \dot{y}) = k_L y + c_L \dot{y}\,, \tag{5.11}$$

where k_L and c_L are the linear stiffness and viscous damping coefficients. The nonlinear term $z(y, \dot{y})$ resembles the dynamic (history-dependent) and hysteretic memory of the system. This term is derived by the first-order differential equation

$$\dot{z}(y, \dot{y}) = \alpha\, \dot{y} - \beta \left(\gamma\, |\dot{y}|\, |z|^{\nu-1}\, z + \delta\, \dot{y}\, |z|^{\nu}\right), \tag{5.12}$$

where the Bouc-Wen parameters α, β, γ, δ, and ν are used to tune the shape and the smoothness of the system's hysteresis loop [94].

The training and validation datasets are each a random phase multisine with $N = 8192$ data points with excited frequencies between $5 - 150\,\text{Hz}$. The sampling frequency is $f_s = 750\,\text{Hz}$. The benchmark provides two test data sets, one with a random phase multisine input ($N = 8192$) and the other with a sine-sweep input ($N = 153\,000$).

Four third-order LMSSN models (the four setups from Fig. 3.3 a)-d)) are trained with LOLIMOT. The SNR is assumed to be $\infty\,\text{dB}$ ($\text{NRMSE}_{\text{noise}} = 0$). A typical convergence curve is shown in Fig. 5.12 of the LMSSN with MISO state equations and affine output equation. It can be seen that training and validation errors are close together, indicating that overfitting is not an issue during training on this process. Moreover, much overfitting is seldom an issue for LMSSN training.

Figure 5.13 shows the two test output signals of the multisine and sine-sweep test datasets. On top are the errors of the BLA and the MISO/affine LMSSN at different complexity stages during training (different number of total splits). The final LMSSN model possesses a total of 74 splits, one in the first state equation, 37 splits in the second state equation, and 36 splits in the third state equation. It is beneficial on this benchmark that the LMSSN can model different state equations at different complexity levels.

Different approaches that have been tested on the Bouc-Wen benchmark are listed in Table 5.4. The LMSSN performs comparable and even better than other state-of-the-art system identification approaches. Note that for the results of the convolutional neural networks and the recurrent neural networks (RNNs) a data record of 153 000 training data points is used. For the here shown LMSSN studies, only 8192 samples were used for training and validation, respectively.

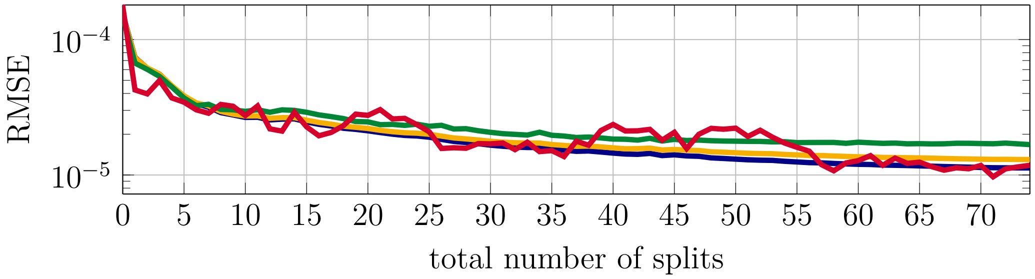


Figure 5.12: Convergence curve of the LMSSN during training on the Bouc-Wen benchmark. Here, (▬) is the convergence on the training data, (▬) on validation data, (▬) on the multisine test data, and (▬) on the sine-sweep test data.

Table 5.4: Comparison of selected methods on Bouc-Wen benchmark [124]

RMSE multisine $\cdot 10^{-5}$	RMSE sinesweep $\cdot 10^{-5}$	Approach (Num. of parameters)	Ref.
PNLSS			
1.34	**1.12**	Decoupled (51)	[35]
1.87	1.20	MIMO / linear $[2-3]$ (90)	[35]
5.42	-	MIMO / linear [2] (34)	[93]
3.15	-	MIMO / linear $[2-4]$ (109)	[93]
1.27	-	MIMO / linear $[2-7]$ (364)	[93]
1.21	-	MIMO / linear [3 5 7] (217)	[93]
Convolutional neural networks			
2.43	1.73	Deep regularized FIR neural network	[83]
3.21	2.97	Deep FIR (without regularization) network	[83]
Recurrent neural networks			
2.80	5.98	LSTM (3 layers)	[123]
7.6	4.1	ReLU RNN	[123]
Other models			
17.0	13.8	NARX with LMN	[8]
16.4	17.2	regularized LMN FIR	[8]
31.2	24.9	OBF with LMN	[8]
7.9	11	Stochastic Subspace	[6]
5.3	1.5	NARX Sigmoidal (1571)	[143]
5.7	1.9	Decoupled NARX (151)	[143]
8.76	6.39	Volterra feedback	[119]
468	18.6	Nelder-Mead	[14]
468	19.0	NOMAD	[14]
LMSSN			
2.24	1.85	MISO / MISO, 76 splits (403)	
1.68	1.18	MISO / affine, 74 splits (393)	
2.31	1.51	MIMO / MISO, 20 splits (153)	
1.56	3.48	MIMO / affine, 43 splits (668)	

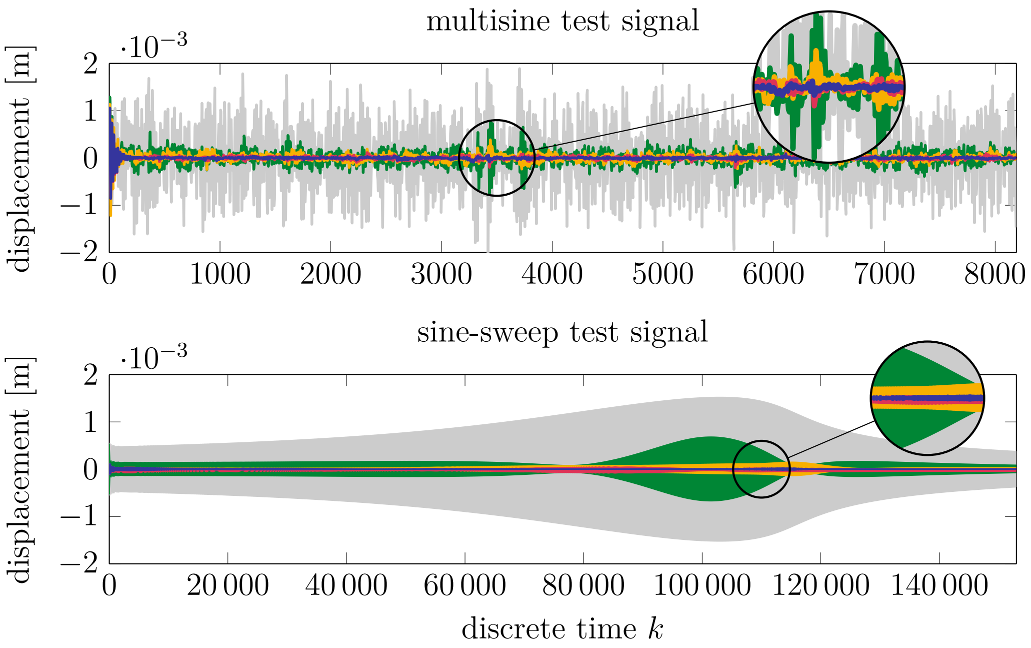


Figure 5.13: Error of LMSSN at different stages during training on the multisine and sine-sweep test signals of the Bouc-Wen benchmark. Here, (⬜) is the test process, (🟩) the error of the BLA, (🟨) the error of LMSSN after the first split, (🟥) error of LMSSN after eight splits, and (🟪) the error of the final LMSSN with 74 splits.

5.6 NO$_x$ Emission Modeling of Turbocharged Diesel Engine

In heavy-duty applications, commonly Diesel engines are employed. One crucial aspect in developing new Diesel engines and their operating strategies is the NO$_x$ emissions produced by the engine. Ever decreasing emissions are demanded by the legislature. The goal to lower the NO$_x$ emissions is thus of utmost importance. To conquer this goal, new control strategies are to be developed which require more accurate emission models of Diesel engines than are currently available.

One way to accomplish more sophisticated and accurate models is by modeling the engines dynamically instead of commonly used static models that only yield acceptable modeling accuracy [49, 4]. An accurate dynamic emissions model might also be employed as a virtual sensor as it is not an easy task to measure NO$_x$ emissions[3], which is only practically feasible on test stands or prototypes due to too excessive

[3] commonly employed NO$_x$-sensors have a tolerance range of round about ±10 % (i.e., Continental UniNO$_x$ sensor [21])

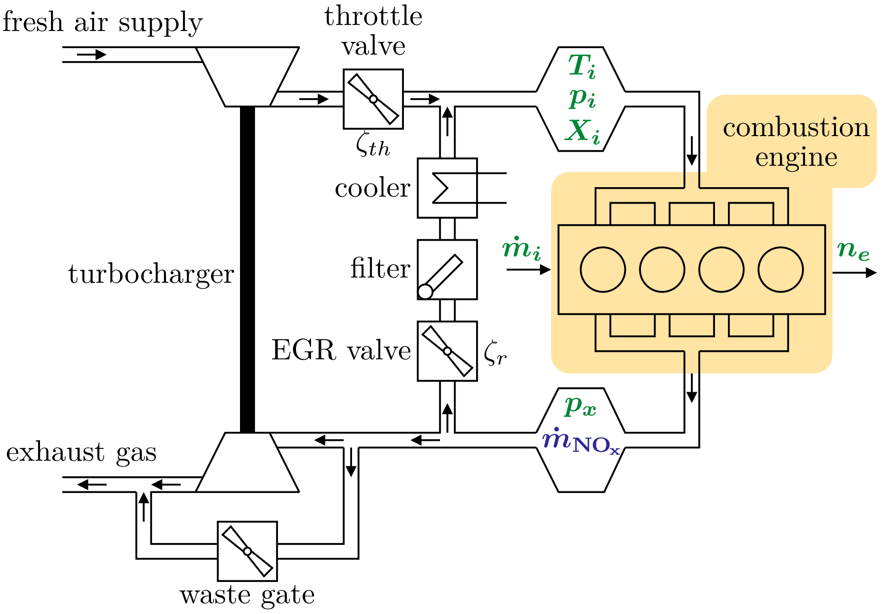


Figure 5.14: Air pathway through a turbocharged Diesel engine with exhaust recirculation (adapted from [149]). The used model inputs for identification are indicated in green, the modeled output (NO_x mass flow) is indicated in blue.

weight, space, and cost. Therefore, a dynamic model would bring the developments of a so-called “digital twin” of the Diesel combustion engine one step closer to realization. A side effect of dynamic modeling is cost reductions. Static measurements (with long hold times until the process settles) are more expensive than dynamic ones, which can be kept shorter in time and require thus less time on test benches.

To show that the LMSSN is an appropriate tool for dynamic $\mathrm{NO_x}$ emissions modeling, a simulation model (GT-Power Engine Simulation) of a 2.9 liters turbocharged direct injection Diesel engine without charge air cooling is used as system under test. This GT-Power model is a solid representation of its physical counterpart [113], which makes the identification of this simulation model with data-driven methods sensible. A schematic sketch of the air pathway of the engine is shown in Fig. 5.14[4].

In the simulation study, the engine is excited by four manipulated variables: engine speed n_e, injection mass flow $\dot{m}_i$, intake throttle valve position ζ_{th}, and the exhaust gas recirculation valve position ζ_r. For excitation of the manipulated variables, a space-filling APRBS is used [131].

Six inputs are used to model the $\mathrm{NO_x}$ mass flow $\dot{m}_{\mathrm{NO}_x}$ (listed in Tab. 5.5). The training signal consists of $N = 20\,700$ samples, the validation dataset of $N = 5980$

[4] Note that not all built-in components are drawn for simplicity.

Table 5.5: Used model inputs and outputs for NO$_x$ modeling

Inputs/Output	**Symbol**	**Description**
$u_1(k)$	n_e	Engine speed
$u_2(k)$	$\dot{m}_i$	Injection mass flow
$u_3(k)$	T_i	Intake temperature
$u_4(k)$	p_i	Intake pressure
$u_5(k)$	p_x	Exhaust pressure
$u_6(k)$	X_i	O$_2$ concentration of intake
$y(k)$	$\dot{m}_{\mathrm{NO_x}}$	NO$_x$ mass flow

data points. For testing, on the one hand, the non-road transient cycle (NRTC) ($N = 12\,327$) is used, which is an official dynamic cycle used by the US EPA and the EU for engine certification [131]. On the other hand, the static model behavior is investigated. For that 1000 space-filling static operating points (Sobol sequence) in the six-dimensional identification input space are held for 6000 time steps and are then concatenated. Only the last model output of each holding period is used for the evaluation of the static behavior (in total 1000 data points). All signals have a sampling frequency of $f_s = 10\,\mathrm{Hz}$.

A second-order LMSSN model is trained with LOLIMOT[5]. For comparison, results from the long short-term memory (LSTM) model and gated recurrent unit (GRU) model have been taken from [131]. Here, the LSTM model consists of two LSTM layers followed by a dense layer with 15 neurons for each layer. The GRU has 35 hidden states followed by a dense layer [131]. The model order of the LMSSN and the structures of the LSTM and GRU have been found by trial and error.

A comparison of the model's training NRMSE, test NRMSE on the NRTC, test NRMSE on the static data, the number of parameters n_θ, and normalized computational demand ε are shown in Fig. 5.15. The normalized computational demand is used to make the models comparable since they have been built in different programming languages. It is the ratio of the model's execution time for one time step to the execution time of 100 consecutive random exponential function evaluations

$$\varepsilon = \frac{1}{N_{\text{test}}} \cdot \frac{t_{\text{model}}}{t_{100\text{exp}}} . \tag{5.13}$$

[5] Training with oblique HILOMOT was tried as well. However, no performance improvements were noted on this process at the cost of a lot longer computation times.

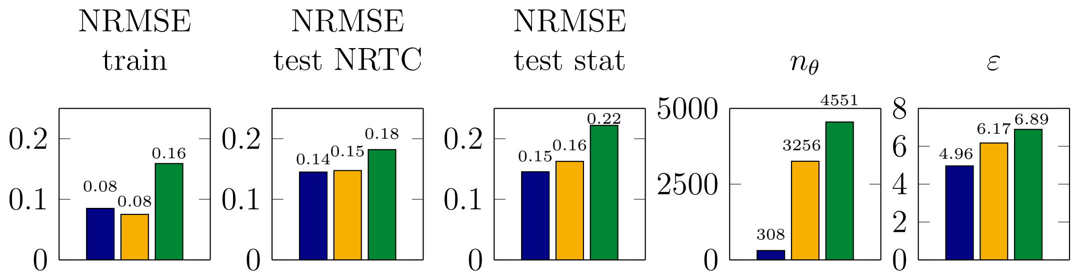


Figure 5.15: Results of the (🟦) LMSSN, (🟧) LSTM model, and (🟩) GRU model for the identification of the NO_x emissions of a GT-Power engine simulation of a 2.9 liters Diesel engine. Here, n_θ is the number of parameters and ϵ is the normalized computational demand. The results of the LSTM model and GRU model are taken from [131].

It can be seen that the LMSSN performs best on the NRTC and static test datasets. Noteworthy is that LMSSN performs best with less than a tenth of parameters compared to the LSTM and GRU model (LMSSN 308 parameters, LSTM 3256 parameters, GRU 4551 parameters). The evaluation time is also the lowest for the LMSSN, but the difference to the other two models is not as large as might be expected from the differences in the number of parameters. This is likely because the calculations of the validity functions for the LMSSN are quite time-consuming. In practice, the low number of parameters and low execution time make the LMSSN model an appropriate candidate to be employed in actual operation on an electronic control unit with limited memory and computation power.

A more detailed look at the model performance of the LMSSN and its dynamic and static behavior can be found in Fig. 5.16. All shown quantities have been normalized to a range between zero and one. First, we focus on the dynamic behavior (upper and lower left plot). Four distinct phases (marked in yellow, green, red, and blue) are investigated that show some interesting features. The yellow phase indicates the transient phase. The LMSSN performs quite well right from the start. This is due to the way the unknown initial state on the test dataset is set. Since it is known for the LMSSN that the state trajectory is inside the unit hypercube during training, the initial state for testing is set to $\hat{\underline{x}}_{0,\text{test}} = 0.5 \cdot \underline{1}^{n_x \times 1}$, which is a viable initialization under the assumption that the test signal operates in roughly the same regime as the training signal.

The LMSSN can approximate the process quite well in the first steady phase (green), while this is not true for the second steady phase (red). As explained in [131], a reason why the LMSSN does not perform well in the second steady phase is likely

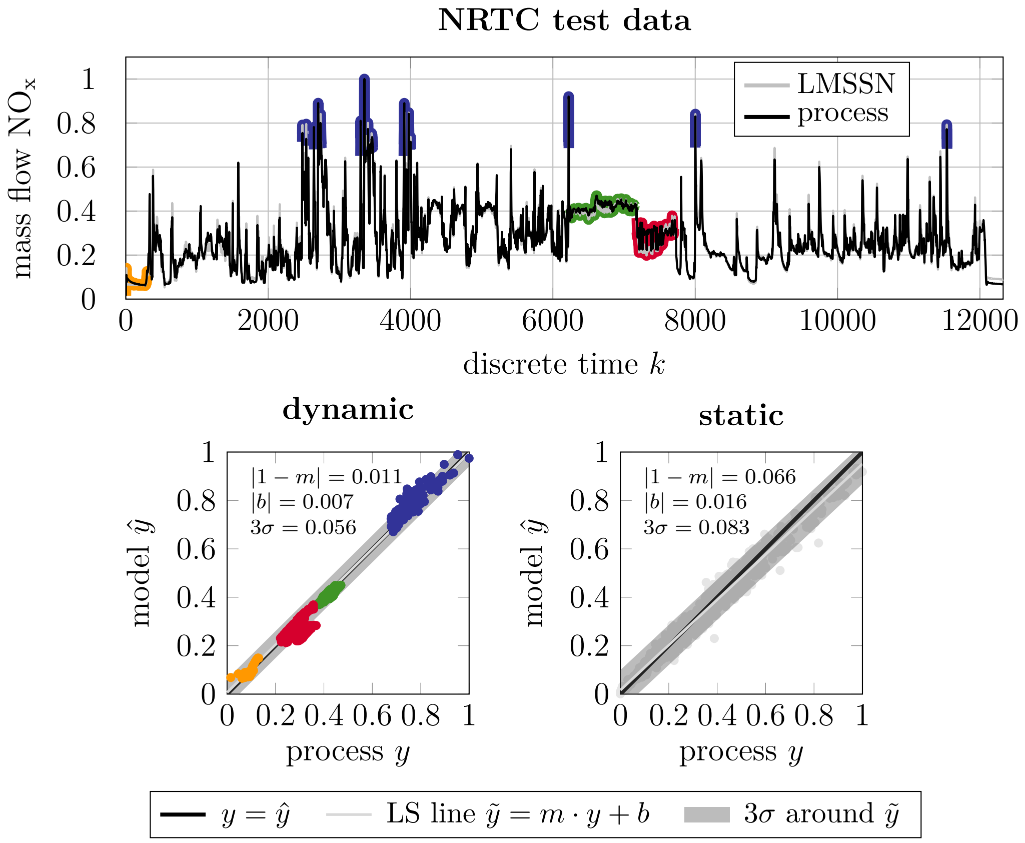


Figure 5.16: Evaluation of the LMSSN on NRTC (upper plot). Correlation plot of the LMSSN model on NRTC test data (lower left plot) and static test dataset (lower right plot). The line (—) $\tilde{y} = m \cdot y + b$ is the least squares regression in the $(y, \hat{y})$-space. An optimal model yields slope $m = 1$, offset $b = 0$, and $\sigma = 0$. For the dynamic data, (●) indicates a transient phase, (●) a first steady phase, (●) a second steady phase, and (●) operating points that lead to high NO_x emissions.

that the GT-Power simulation model is flawed in this region as also other models are congruently not able to model this sequence. The blue phases indicate operating regions of high NO_x emissions. Here, the LMSSN is also able to approximate the process well.

Second, the static behavior of the LMSSN is shown in the lower right plot of Fig. 5.16. Here, the correlation plot between the static process and model output is shown. The least squares regression line slightly underestimates the static behavior for high amplitudes but otherwise yields good results.

5.7 Subprocess Modeling of Combine Harvester

Combine harvesters are complex agricultural machines that automate the harvesting process of various kinds of grain crops to a large proportion. Due to the enormous complexity of combine harvesters, it is common to model combine harvesters as “the sum of its subprocesses”. The main subprocesses are threshing, separation, and cleaning [23].

After cutting, the crop enters the threshing process (see Fig. 5.17). Here, the separation of grain kernels and chaff from the straw takes place. Two material flows exit the threshing process and enter the separation and cleaning subprocesses, respectively. In the separation process, the remains of grain kernels left in the straw are once more tried to be separated from the straw. The separated grain and the other material flow from the threshing enter the cleaning subprocess, where the grain kernels are separated from the chaff and remaining straw pieces [62].

To increase the efficiency and achieve optimal performance of the combine harvester, accurate models of the subprocesses are needed. Experiments have shown that harvesting is a nonlinear dynamic process [81] requiring sophisticated nonlinear dynamic models.

One subprocess shall be modeled with the use of LMSSN. The chosen subprocess is threshing and how this process influences the losses after separation, see Fig. 5.17. The crop passes through a gap between the threshing drum and threshing concave, which is variable in height. The throughput (u_1; the amount of passing crop), the height of the gap (u_2), and the rotational speed of the drum (u_3) influence the threshing process most dominantly.

Two datasets (the first for estimation with $N = 4527$ data points and the second

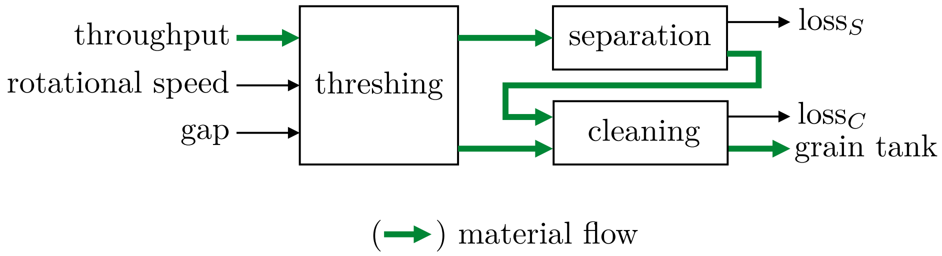


Figure 5.17: Simplified block diagram of threshing process within a combine harvester (adapted from [62])

Table 5.6: Results on test data for modeling the separation losses within a combine harvester.

NRMSE	Approach
0.56	BLA
0.52	Static LOLIMOT
0.38	LOLIMOT NARX ($m = 1$)
0.36	LOLIMOT NARX ($m = 2$)
0.26	LMSSN ($n_x = 1$)

for testing with $N = 2234$ data points) are available. Both datasets are obtained from the field operation of a combine harvester, where the three inputs are excited by an optimized nonlinear input signal (OMNIPUS) [62]. The datasets have been preprocessed to eliminate dead times. Since the throughput is believed to have the strongest influence on the process, a single-input single-output (SISO) LMSSN will be trained. Note that all results are normalized.

In total, four different LMSSNs were trained where all state equations are modeled by one LMN and the output equation is modeled by another LMN. LMSSN models from first to fourth order are tried. It turns out that an increasing dynamic order of the LMSSN does not yield better performance on validation data.

The first-order LMSSN achieves an NRMSE on test data of 0.26 (one LM in the state equation, two LMs in the output equation). Therefore, the resulting LMSSN structure resembles a Wiener block-oriented model (see Sect. 3.5.2).

Employing a purely static model (static LOLIMOT model trained with the LMN-TOOL v1.5.2 [48]) yields an NRMSE of 0.52, while the BLA (LTI model) achieves an NRMSE of 0.56. Only the combined use of a dynamic *and* nonlinear model in the form of the LMSSN makes the test error drop roughly to half (NRMSE = 0.26), compared to the nonlinear static and linear dynamic model (see Tab. 5.6). The normalized process, LMSSN model, and BLA output on test data can be seen in Fig. 5.18.

The output equation of the first-order LMSSN model is shown in Fig. 5.19. The split in the $\hat{x}_1(k)$-dimension moved outside the normal range $[0, 1]$ (the black plane in left plot), which produces seemingly exponential behavior in the interpolation range $[0, 1]^2$ (right plot). The LMSSN is able to perform an automatic split adaption through the rescaling of the state trajectory after every split (see Sect. 3.3.3). This

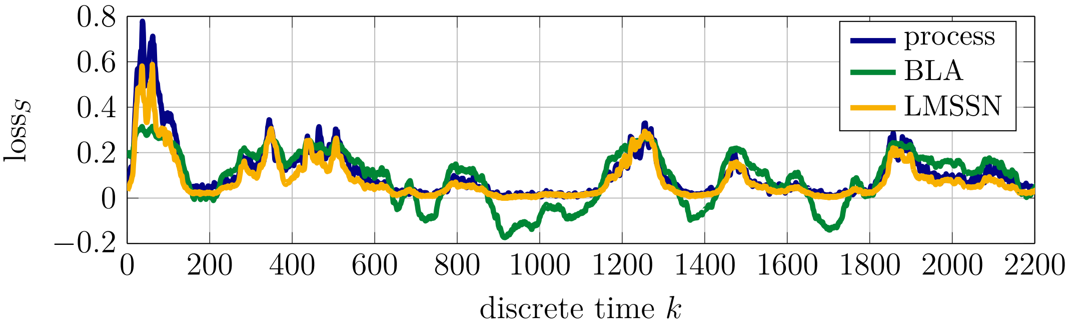


Figure 5.18: Normalized separation losses of process, BLA, and LMSSN for modeling of the threshing process of a combine harvester.

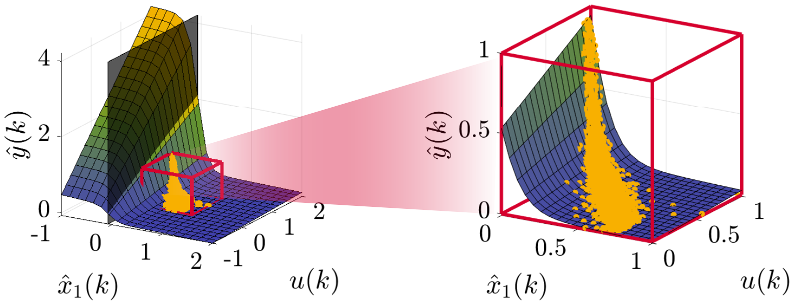


Figure 5.19: Normalized output equation $\hat{y}(k) = g(\hat{x}_1(k), u(k))$ of a first-order LMSSN with two LMs. Left plot: the split (black plane) moved outside the interval $[0, 1]$ in the $\hat{x}_1(k)$-dimension to $\hat{x}_1(k) = -0.073$. Right plot: this produces in the interpolation range $[0, 1]^2$ seemingly exponential behavior.

automatic split adaption leads to the ability of LMSSN to model, for example, exponential behavior with a lot fewer LMs than tree-construction algorithms without split adaption. This is the case, as usually for LMNs, the overall nonlinear behavior is dominantly defined by the LMs. In the case shown here, though, the split adaption mechanism mainly utilizes the shape of the validity functions instead.

6 Conclusion and Outlook

This thesis focuses on developing innovative methods in the field of nonlinear system identification.

Conclusion

A novel class of model structures and associated training algorithms for building data-driven nonlinear state space models is developed.

To achieve this goal, the foundations of system identification have been explained. This encompasses different dynamics realizations such as internal and external dynamics approaches for which a non-comprehensive overview of advantages and disadvantages is given. Neural networks with an emphasis on local model networks (LMNs) as well as different nonlinear optimization algorithms are covered. For recurrent structures like the nonlinear state space model, the problem of backpropagation-through-time is explained. Special attention is then given to different training strategies regarding the gradient update frequency. Here, the cases of batch, mini batch, and sample adaption training are studied for which different options for the state initialization are introduced. The different gradient update strategies are then assessed regarding their stochasticity, computational demand, the dimensionality of the optimization problem, and their likeliness for error-prone gradients. Closing the foundations chapter, the bias/variance tradeoff and model complexity selection is explained.

The novel local model state space network (LMSSN) is developed. The model structure of LMSSN can be explained from a neural network and a system identification perspective. The local linear model tree (LOLIMOT) and hierarchical local model tree (HILOMOT) algorithms are extended to be applicable to state space. This concerns a strategy for initialization, transforming the state trajectory and adapting splits, constructing local loss functions in the state equation, and the notion of a worst local model (LM) is expanded to multiple LMNs. Additionally, data-driven local coordinates (DDLC) are covered and how this strategy is employed during optimization. The model's favorable extrapolation behavior is explained. Finally, some numerical issues with normalized radial basis function (NRBF) validity functions in recurrent structures are explained and it is explained how hierarchical sigmoid functions as validity functions, for which the partition of unity also holds, can alleviate those problems.

Common recurrent neural networks (RNNs) are similar but not equal to nonlinear state space models and the conversion from one model to the other is shown. Different building blocks for deep RNNs are explained from a controls perspective, including the simple recurrent, fully connected, long short-term memory (LSTM), and gated recurrent unit (GRU) layer. A comprehensive case study compares different deep RNN structures and closes this part.

The overall very good performance of LMSSN is demonstrated in various studies and applications. Studies regarding the partitioning strategy of LMSSN and regarding different dynamics realizations are carried out on two different artificial test setups. Challenging nonlinear system identification benchmarks are addressed. The LMSSN's performance is comparable and sometimes beyond the performance of other state-of-the-art nonlinear system identification algorithms. It is shown that the LMSSN method can handle a very wide variety of processes and consistently computes expressive yet compact models. The LMSSN is successfully applied to two real-world processes. The first process deals with the $\mathrm{NO_x}$ modeling of a turbocharged Diesel engine. It is shown that the LMSSN achieves superior test performance in comparison to other identification algorithms while having the lowest model complexity in terms of number of parameters and model order. The second real-world application is the subprocess modeling of a combine harvester. Here, the threshing process is modeled. The LMSSN best describes the process in resemblance to a Wiener block-oriented model.

Outlook

The following points can be addressed in future research.

Imposing Structure Through Nonlinear Canonical Representations There exist different canonical representations for linear state space models, such as the observable, controllable, or Jordan canonical forms. This removes the redundancy of the state space model but imposes severe restrictions concerning the model's flexibility. Some work has been done regarding the extension of the linear canonical forms into the nonlinear world [151]. The properties of such canonical forms in the nonlinear case and incorporation possibilities into LMSSN shall be further investigated. These restriction are assumed to be too severe, at least in the pure black-box case. More promising seems to pursue fully populated matrices/vectors but combined with L_1-norm regularization. L_1-norm regularization can generate sparse representations (many parameters are driven to zero) but keeps the structure flexible.

Local Optimization As stated in [85], global optimization runs the risk of locally inaccurate LMs (meaning that they do not resemble appropriate linearizations of the nonlinear system in the according operating point) for the sake of overall lower cost function values. Therefore, for static and external dynamics LMNs, regularization via local estimation is very commonly utilized. The idea is to estimate the parameters of each LM separately, ignoring/neglecting their coupling due to the validity functions' overlap. Since a single loss function measures the mismatch between process and model output for LMSSNs, the parameter estimation becomes intrinsically global. However, in [153], it is demonstrated how local and global estimation can be combined and what benefits can be obtained by this unification. This idea and its transfer to the LMSSN is quite promising. It can be realized by adding the local losses $I_{i,j}^{[s]}$ and $I_m^{[o]}$ from (3.15) and (3.16), respectively, to the global loss I_{global} in the objective to be optimized. This can be done in a weighted manner, i.e., $I_{\text{total}} = \alpha I_{\text{global}} + (1-\alpha)\left(\sum_i \sum_j I_{i,j}^{[s]} + \sum_m I_m^{[o]}\right)$. This promises to exploit all benefits from local estimation without reducing the computational load, though.

Excitation Signals Based on State-space-filling Designs For all data-driven modeling approaches, the modeling quality highly depends on the information contained in the data that is used for training. For external dynamics approaches, much work has been done that aims for optimized nonlinear input signals (OMNIPUSs) [52, 62]. The idea is to build in a first step a rough proxy model of the process for which the input signal shall be designed. In a second step, the input signal is designed so that it covers the input space to the proxy model in a space-filling manner. Currently, external dynamics proxy models are employed. Here, the idea for future research is to employ state space proxy models instead and apply space-filling designs to the inner input space $\underline{\tilde{u}}(k) = [\underline{\hat{x}}^T(k)\ u(k)]^T$.

A Background and Notation

A.1 Neural Networks

Let us consider the basis function formulation

$$\hat{y} = \sum_{j=1}^{n_m} \theta_j \mathit{\Phi}_j \left(\underline{u}, \underline{\theta}_j^{[nl]} \right) . \tag{A.1}$$

The model output $\hat{y}$ is calculated as a weighted sum of n_m basis functions $\mathit{\Phi}_j(\cdot)$. The basis functions are weighted with the linear parameters θ_j and depend on the input vector $\underline{u}$ and a set of nonlinear parameters $\underline{\theta}_j^{[nl]}$. Now we say that the input arguments to the basis functions are combined into z_j, which are linear combinations of the inputs (with an additional offset)

$$\mathit{\Phi}_j \left(\underline{u}, \underline{\theta}_j^{[nl]} \right) = \mathit{\Phi}_j \left(z_j(\underline{u}) \right) \qquad \text{with} \quad z_j(\underline{u}) = \underline{w}_j^T \underline{u} + b_j \, . \tag{A.2}$$

Note that weight-vector $\underline{w}_j^T$ and offset b_j are, in this case, the hidden layer parameters $\underline{\theta}_j^{[nl]}$ from (A.1) and $\mathit{\Phi}_j$ is called a *neuron* of the neural network.

Different well-known neural network types are obtained depending on the choice of $\mathit{\Phi}_j(\cdot)$. The multilayer perceptron (MLP) network is obtained if a saturation function like the sigmoid function

$$\sigma_j(\underline{u}) = \frac{1}{1 + \exp(-z_j(\underline{u}))} \tag{A.3}$$

or the tanh(·) is employed. The name originates from the early works on the perceptron of Rosenblatt [108].

Another popular choice are rectified linear unit (ReLU) functions

$$\mathrm{ReLU}_j(\underline{u}) = \max(0, z_j(\underline{u})) \, . \tag{A.4}$$

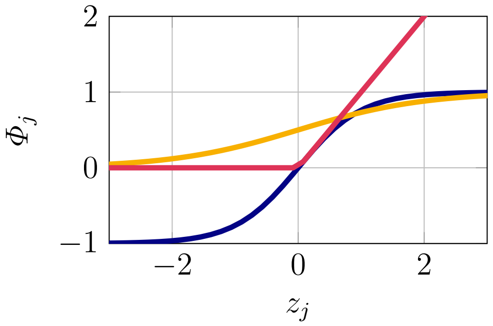


Figure A.1: Illustration of different activation functions Φ_j. Shown are (▬) the tanh, (▬) sigmoid, and (▬) ReLU activation.

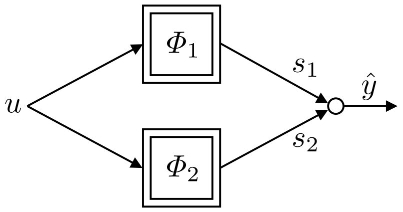


Figure A.2: Network setup

Hidden layers with those ReLU functions are particularly popular in deep learning applications. The layers of these neurons are oftentimes referred to as fully-connected or dense layers.

Those three nonlinear functions are shown in Fig. A.1. They can be interpreted as *activation* functions as there are parts where the nonlinear function lets an input pass and where the input is blocked (or, in the case of the tanh(·), set to −1).

A fourth-order polynomial

$$y = 0.5 + 0.02u + 0.055u^2 + 0.02u^3 - 0.03u^4 \tag{A.5}$$

is considered as a small illustrative example. This function is to be approximated with a neural network (one hidden layer with two neurons), as depicted in Fig. A.2. The model output $\hat{y}$ is calculated by

$$\hat{y} = s_1 + s_2 = \theta_1 \cdot \Phi_1(\underbrace{w_1 u + b_1}_{z_1}) + \theta_2 \cdot \Phi_2(\underbrace{w_2 u + b_2}_{z_2}), \tag{A.6}$$

with the six trainable parameters $\theta_1, w_1, b_1, \theta_2, w_2, b_2$. Three neural networks (one with tanh(·), one with sigmoid, and the third with ReLU activation functions) are trained with a Quasi-Newton optimizer with $N = 50$ data points which are, for

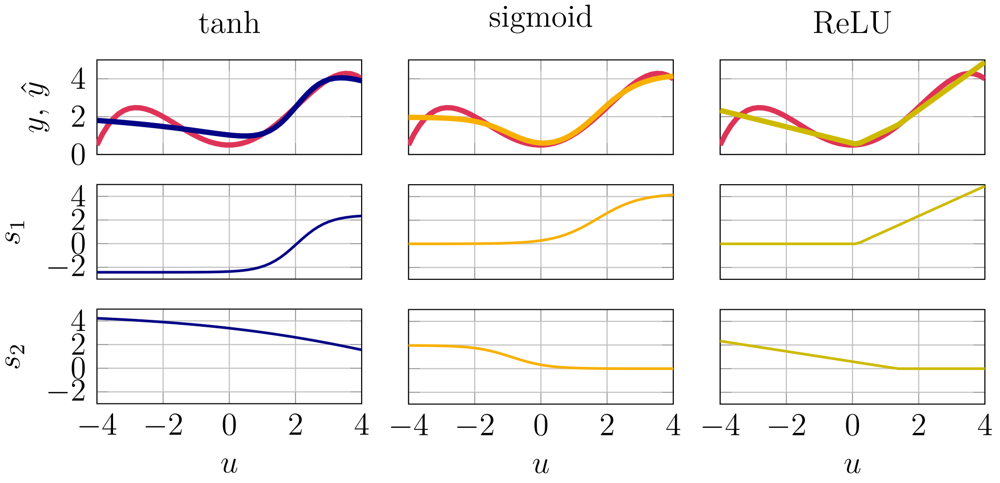


Figure A.3: Neural networks with one hidden layer with two neurons. Here, (▬) is the fourth-order polynomial process. The model output $\hat{y}$ is the sum of the two scaled neuron outputs $\hat{y} = s_1 + s_2$, shown by (▬), (▬), and (▬) for different activation functions.

simplicity, linearly spaced points in the interval $[-4, 4]$. The results are shown in Fig. A.3. It can be seen how the different activation functions are active in a part of the input space and inactive in the other part. A superposition of an increasing number of nodes makes neural networks very expressive.

A.2 Comparison of NRBF Network and LMN

A visual example of a normalized radial basis function (NRBF) network in contrast to an local model network (LMN) is shown in Fig. A.4 for the one-dimensional case. Shown are an NRBF network (left plots, blue) and an LMN (right plots, yellow) with NRBF validity functions with three weighted nodes / local models for each neural network (plots s_1-s_3). The location and standard deviations of the radial basis functions (RBFs) are chosen for both neural networks to $(\mu_1, \sigma_1) = (-3, 0.5)$, $(\mu_2, \sigma_2) = (-1, 0.5)$, and $(\mu_3, \sigma_3) = (2, 0.5)$.

The local characteristics of both networks are clearly visible. The additional parameter per node / local model gives the LMN greater modeling capabilities when the number of local models (LMs) is equal for both models. In the case of an infinitely small standard deviation ($\sigma \to 0$), the NRBF network is hard-switching its constant local behavior depending on the location in the input space. The same goes for the LMN as it switches between different affine functions, also depending on the location

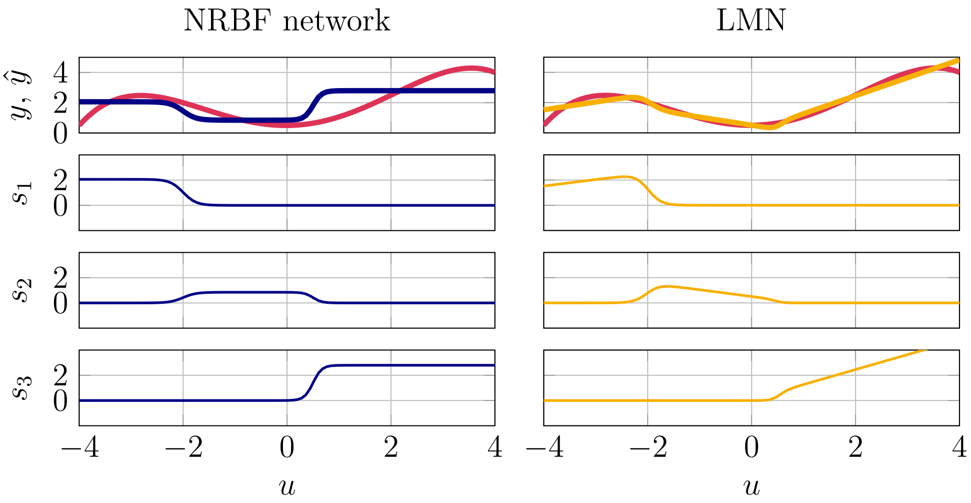


Figure A.4: Comparison of an NRBF network with an LMN. Here, (🟥) is the true process. The model output $\hat{y}$ is the sum of the three (🟦) weighted neurons / (🟧) local models as $\hat{y} = s_1 + s_2 + s_3$.

in the input space. In this extreme case ($\sigma = 0$), an LMN is equivalent to a piecewise affine (PWA) model.

A.3 Nonlinear Optimization Algorithms

Some nonlinear optimization algorithms are explained in more detail.

Newton's Method For the steepest descent method, only the gradient $\underline{g}(i)$ is necessary for the parameter update. In contrast, Newton's method (and other second-order methods) use information about the curvature of the loss function in addition to the gradient for the parameter updates [3]. A quadratic approximation of the loss function around an initial parameter vector $\underline{\theta}(i)$ can be written as

$$I(\underline{\theta}(i+1)) \approx I(\underline{\theta}(i)) + (\underline{\theta}(i+1) - \underline{\theta}(i))^T \underline{g}(i) + \frac{1}{2}(\underline{\theta}(i+1) - \underline{\theta}(i))^T \underline{H}(i)(\underline{\theta}(i+1) - \underline{\theta}(i)) \,, \tag{A.7}$$

where $\underline{g}(i)$ is the gradient and $\underline{H}(i)$ the Hessian matrix of the loss function with respect to $\underline{\theta}(i)$. Now, one can set the gradient of the approximate loss function regarding $\underline{\theta}(i+1)$ (A.7) to $\underline{0}$, leading to the optimality condition

$$\frac{\partial I(\underline{\theta}(i+1))}{\partial \underline{\theta}(i+1)} \overset{!}{=} \underline{0} \,, \tag{A.8}$$

which gives the condition in terms of the quadratic approximation as

$$\underline{g}(i) + \underline{H}(i)(\underline{\theta}(i+1) - \underline{\theta}(i)) \overset{!}{=} \underline{0}\,. \tag{A.9}$$

After rearranging (A.9), the following parameter update equation is obtained

$$\hat{\underline{\theta}}(i+1) = \underline{\theta}(i) - \underline{H}(i)^{-1}\underline{g}(i)\,. \tag{A.10}$$

Here, the $\hat{(\cdot)}$ denotes that the obtained parameters are optimal with respect to the quadratic approximation of the loss function. Comparing (2.56) with (A.10) shows that for the classical Newton method $\eta(i) = 1$ and $\underline{R}(i) = \underline{H}^{-1}(i)$.

A step size is reintroduced since one can never really know how well the quadratic approximation fits the actual loss function surface. It is most of the time optimized via line search. This then yields the damped Newton update equation as

$$\underline{\theta}(i+1) = \underline{\theta}(i) - \eta(i)\underline{H}(i)^{-1}\underline{g}(i)\,. \tag{A.11}$$

A major concern with Newton's method is that the updated parameters will only decrease the loss function value for positive definite Hessians $\underline{H}(i)$. This is surely true in the vicinity of an optimum but is not necessarily true for an initial parameter vector. Modified Newton methods can therefore be employed, using an altered Hessian $\tilde{\underline{H}}(i)$, which is close to the original $\underline{H}(i)$, but it is ensured that this matrix is positive definite [89].

The most important advantage (of using $\underline{H}(i)$ in the parameter update) is that the rate of convergence is of second order. However, this comes at the cost that second-order derivatives have to be computed and a computationally expensive matrix inversion needs to be performed.

Some properties of Newton's method are listed below [89]:

- requires second-order derivatives
- cubic computational complexity due to matrix inversion
- quadratic memory requirement complexity (storage of Hessian)
- fastest convergence normally encountered in nonlinear optimization
- requires a single iteration for the solution of a linear regression problem
- unaffected by a linear transformation of the parameters
- suited for small problems (order of 10 parameters).

Quasi-Newton Method Since the calculation of the Hessian $\underline{H}(i)$ is computationally expensive, Quasi-Newton methods use only information of loss function evaluations $I(\underline{\theta}(i))$ and the gradient $\underline{g}(i)$ to approximate the Hessian $\underline{H}(i)$. This can be done with the formula by Broyden [13], Fletcher [36], Goldfarb [42], and Shanno [127] (BFGS), as it has been proven to be the most effective general purpose method. For a detailed account of the calculation, refer to [112].

Again, some properties [89]:

- no requirement of second-order derivatives
- quadratic computational complexity owing to matrix multiplication
- quadratic memory requirement
- very fast convergence
- requires at most n_θ iterations for the solution of a linear regression problem
- affected by a linear transformation of the parameters
- suited best for medium sized problems (order of 100 parameters).

Adaptive Moment Estimation Method Another widespread choice for nonlinear optimization is the adaptive moment estimation (ADAM) method [59]. ADAM has been applied successfully on various machine learning tasks [148, 59] and is intensively studied in the machine learning community [104, 67]. ADAM computes individual adaptive learning rates for different parameters from estimates of the gradients' first- and second-order moments. It was designed to combine the advantages of two other optimization algorithms, the adaptive gradient algorithm (ADAGRAD) (which works well with sparse gradients) and root mean square propagation (RMSprop) (which works well in online settings). Having the advantageous properties of both of these algorithms enables us to use ADAM for a broader range of tasks. ADAM can also be looked at as the combination of RMSprop and stochastic gradient descent (SGD) with momentum. It hereby combines an exponentially decaying average of past gradients $\underline{m}(i)$ and an exponentially decaying average of past squared gradients $\underline{v}(i)$

$$\underline{m}(i) = \beta_1 \underline{m}(i-1) + (1-\beta_1)\,\underline{g}(i)\,, \tag{A.12}$$

$$\underline{v}(i) = \beta_2 \underline{v}(i-1) + (1-\beta_2)\,\underline{g}^2(i)\,. \tag{A.13}$$

The squared gradients are calculated element-wise (with the Hadamard product) as $\underline{g}^2(i) = \underline{g}(i) \odot \underline{g}(i)$. Here, $\underline{m}(i)$ and $\underline{v}(i)$ are estimates for the first-order moment (the mean) and the second-order moment (the uncentered variance), hence the name adaptive moment estimation.

At the beginning of the training, $\underline{m}(i)$ and $\underline{v}(i)$ are initialized as zero-vectors. Therefore, they are biased towards zero in the early epochs. These biases are counteracted by computing bias-corrected first- and second-order moment estimates as

$$\hat{\underline{m}}(i) = \frac{\underline{m}(i)}{1 - \beta_1^i}\,, \tag{A.14}$$

$$\hat{\underline{v}}(i) = \frac{\underline{v}(i)}{1 - \beta_2^i}\,. \tag{A.15}$$

For the derivation, refer to the original paper [59]. The parameter update rule is eventually calculated as

$$\underline{\theta}(i+1) = \underline{\theta}(i) - \frac{\eta}{\sqrt{\hat{\underline{v}}(i)} + \epsilon}\hat{\underline{m}}(i)\,. \tag{A.16}$$

The authors propose standard values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ.

Some properties of the ADAM optimizer [59, 16]:

- no requirement of second-order derivatives
- linear computational complexity
- linear memory requirement complexity
- the actual step size taken by ADAM in each iteration is approximately bounded by the step size hyperparameter. This property adds intuitive understanding compared to other unintuitive learning rate hyperparameters in common machine learning optimizers.
- step size of ADAM is invariant to the magnitude of the gradient.

A.4 Levenberg-Marquardt Algorithm Implementation

In the following, some details regarding the Levenberg-Marquardt algorithm and its implementation is covered. The general procedure is adapted from [96]. To solve (2.72), it is written as

$$\underline{\theta}^* = \underline{\theta} - \eta(\underline{J}^T\underline{J} + \lambda^2\underline{I})^{-1}\underline{J}^T\underline{e}\,, \tag{A.17}$$

where the updated parameter vector is denoted by $(\cdot)^*$ and all indices are left out for brevity. Now, we rewrite the problem according to

$$\left(\underline{J}^T\underline{J} + \lambda^2\underline{I}\right)\underline{p} = \underline{J}^T\underline{e} \tag{A.18}$$

and solve this system of equations (also called the normal equation), where the parameter update $\underline{p}$ is defined as

$$\underline{p} = -\frac{\underline{\theta}^* - \underline{\theta}}{\eta}\,. \tag{A.19}$$

The singular value decomposition (SVD) of $\underline{J}$ is given by

$$\underline{J} = \underline{U}\,\underline{\Sigma}\,\underline{V}^T\,. \tag{A.20}$$

Since we usually have $N \geq n_\theta$, the thin, or economy, SVD is used, which means that $\underline{U}$ only contains n_θ columns and $\underline{\Sigma}$ is a square matrix $\in \mathbb{R}^{n_\theta \times n_\theta}$. When $\underline{J}$ is singular, $\underline{J}$ has rank $\tilde{n}_\theta \leq n_\theta$ and the truncated SVD is given by

$$\underline{J} = \underline{U}\,\mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_{\tilde{n}_\theta}, 0, \ldots, 0)\underline{V}^T\,. \tag{A.21}$$

Using $\left(\underline{U}\,\underline{\Sigma}\,\underline{V}^T\right)^T = \underline{V}\,\underline{\Sigma}^T\underline{U}^T$ and $\underline{U}^T\underline{U} = \underline{I}$ (since $\underline{U}$ is a unitary matrix) yields

$$\left(\underline{V}\,\underline{\Sigma}^T\underline{\Sigma}\,\underline{V}^T + \lambda^2\underline{I}\right)\underline{p} = \underline{V}\,\underline{\Sigma}^T\underline{U}^T\underline{e}\,. \tag{A.22}$$

Now we can define a matrix $\underline{\Gamma}$, which adds the scaled identity matrix $\lambda^2\underline{I}$ to $\underline{\Sigma}^T\underline{\Sigma}$, which is an $(n_\theta \times n_\theta)$ matrix as

$$\underline{\Gamma} = \mathrm{diag}\left(\sigma_1^2 + \lambda^2, \sigma_2^2 + \lambda^2, \ldots, \sigma_{\tilde{n}_\theta}^2 + \lambda^2, \lambda^2, \ldots, \lambda^2\right)\,. \tag{A.23}$$

Using the property $\underline{V}^T\underline{V} = \underline{I}$ (since $\underline{V}$ is also a unitary matrix), inserting $\underline{\Gamma}$ and rearranging (A.22) yields

$$\underline{p} = \underline{V}\,\underline{\Gamma}^{-1}\underline{\Sigma}^T\underline{U}^T\underline{e}\,. \tag{A.24}$$

At last, let us define $\underline{\Lambda}$ as

$$\underline{\Lambda} = \underline{\Gamma}^{-1}\underline{\Sigma}^T = \mathrm{diag}\left(\frac{\sigma_1}{\sigma_1^2 + \lambda^2}, \frac{\sigma_2}{\sigma_2^2 + \lambda^2}, \ldots, \frac{\sigma_{\tilde{n}_\theta}}{\sigma_{\tilde{n}_{\tilde{\theta}}}^2 + \lambda^2}, 0, \ldots, 0\right)\,, \tag{A.25}$$

which then yields the final equation

$$\underline{p} = \underline{V}\,\underline{\Lambda}\,\underline{U}^T\underline{e}\,. \tag{A.26}$$

Before computation of the SVD, the columns of $\underline{J}$ are normalized with respect to their root mean squared (rms) value for better conditioning of the SVD. Vector $\underline{p}$ is afterward "denormalized" accordingly. The learning rate is chosen as $\eta = 1$ so that operation is purely controlled by λ. The factor λ is initialized with the largest singular value in the first iteration σ_1 [37]. The updated λ, denoted by λ^*, is calculated as

$$\lambda^* = \begin{cases} \frac{\lambda}{2}, & \text{if } I(\underline{\theta}^*) < I(\underline{\theta}) \\ \lambda \cdot \sqrt{10}, & \text{if } I(\underline{\theta}^*) \geq I(\underline{\theta}) \end{cases}. \tag{A.27}$$

B LMSSN Details

B.1 LMSSN Equations

In the following, the local model state space network (LMSSN) notation is explained. Let us consider the multiple-input single-output (MISO) LMSSN. Note that an extension to the multiple-input multiple-output (MIMO) case is straightforward.

The MISO LMSSN can be written as

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \sum_{j=1}^{\max(n_{m_i},n_o)} \Big(\underline{o}_j + \underline{A}_j \underline{\hat{x}}(k) + \underline{B}_j \underline{u}(k)\Big)\, \Phi_j^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \hat{y}(k) &= \sum_{j=1}^{\max(n_{m_i},n_o)} \Big(p_j + \underline{c}_j^T \underline{\hat{x}}(k) + \underline{d}_j^T \underline{u}(k)\Big)\, \Phi_j^{[o]}(\underline{\hat{x}}(k), \underline{u}(k))\,. \end{aligned} \tag{B.1}$$

The parameters $\underline{o}_j, \underline{A}_j, \underline{B}_j, p_j, \underline{c}_j^T, \underline{d}_j^T$ stand for one “slice” in the *third* dimension of Fig. 3.10 as

$$\underline{\Theta}_j = \begin{bmatrix} \underline{o}_j & \underline{A}_j & \underline{B}_j \\ p_j & \underline{c}_j^T & \underline{d}_j^T \end{bmatrix}\,. \tag{B.2}$$

The first “slice” $\underline{\Theta}_1$ is fully populated since we always start with an affine approximation. In all other “slices” ($j = 2, \ldots, \max(n_{m_i}, n_o)$), some rows may contain parameters, while others do not, depending on the way each individual state or output equation is split (see for example Fig. 3.10, where for $j = 2$ the third row has no entries and for $j = 3$ entries are missing in all rows except the second). Note that $\max(n_{m_i}, n_o)$ is used for ease of notation without loss of generality as not all rows of each $\underline{\Theta}_j$ may contain parameters. This makes all combinations of MIMO and MISO local model networks (LMNs) and affine functions in state and output equation possible.

The validity functions, from now on in short

$$\underline{\Phi}_j(k) = \begin{bmatrix} \underline{\Phi}_j^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \Phi_j^{[o]}(\underline{\hat{x}}(k), \underline{u}(k)) \end{bmatrix}, \tag{B.3}$$

are constructed either by normalized radial basis functions (NRBFs) or by hierarchical sigmoid functions. Just as in (B.2), the first "slice" $\underline{\Phi}_1(k)$ is fully populated, as in all other "slices" ($j = 2, \ldots, \max(n_{m_i}, n_o)$) some rows may contain parameters, while others do not, depending on the way each individual state or output equation is split.

Validity Functions – Example 1 For example, consider a second-order LMSSN with a MIMO LMN with three local models (LMs) for the state equation and with an affine output equation. Then, (B.3) becomes for $j = 1, 2, 3$

$$\underline{\Phi}_1(k) = \begin{bmatrix} \Phi_1^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \Phi_1^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ 1 \end{bmatrix} \quad \underline{\Phi}_2(k) = \begin{bmatrix} \Phi_2^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \Phi_2^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ 0 \end{bmatrix} \quad \underline{\Phi}_3(k) = \begin{bmatrix} \Phi_3^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \Phi_3^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ 0 \end{bmatrix}. \tag{B.4}$$

Note that the first two rows of $\underline{\Phi}_1(k)$, $\underline{\Phi}_2(k)$, and $\underline{\Phi}_3(k)$ contain the same values, respectively, as the validity function is shared for the MIMO LMN.

Validity Functions – Example 2 As another example, consider a second-order LMSSN with a MISO LMN with two LMs for the first state equation, and a MISO LMN with three LMs for the second state equation, and an affine output equation. Then, (B.3) becomes for $j = 1, 2, 3$

$$\underline{\Phi}_1(k) = \begin{bmatrix} \Phi_{1,1}^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \Phi_{2,1}^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ 1 \end{bmatrix} \quad \underline{\Phi}_2(k) = \begin{bmatrix} \Phi_{1,2}^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ \Phi_{2,2}^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ 0 \end{bmatrix} \quad \underline{\Phi}_3(k) = \begin{bmatrix} 0 \\ \Phi_{2,3}^{[s]}(\underline{\hat{x}}(k), \underline{u}(k)) \\ 0 \end{bmatrix}. \tag{B.5}$$

Note that in $\Phi_{i,j}^{[s]}$, the first index indicates the i-th state equation and the second index stands for the j-th local model.

Normalized Radial Basis Functions The NRBF validity functions are calculated by

$$\underline{\Phi}_j(k) = \underline{\Psi}_j(k) \oslash \|\underline{\Psi}(k)\|_1, \tag{B.6}$$

where $\underline{\Psi}_j(k)$ denote the radial basis functions (RBFs), $\oslash$ element-wise division, and $\|\underline{\Psi}(k)\|_1$ denotes a vector with the 1-norms of the RBFs in each state and output equation, which is calculated by $\sum_{j=1}^{\max(n_{m_i},n_o)} \underline{\Psi}_j(k)$ with $\underline{\Psi}_j(k) \geq \underline{0}$. For the divisions, a very small ϵ is added to the denominator.

Normalized Radial Basis Functions – Example 1 For example, consider again a second-order LMSSN with a MIMO LMN with three LMs for the state equation and with an affine output equation. Then, (B.6) becomes

$$\underline{\Phi}_1(k) = \begin{bmatrix} \frac{\Psi_1^{[s]}}{\Psi_1^{[s]}+\Psi_2^{[s]}+\Psi_3^{[s]}+\epsilon} \\ \frac{\Psi_1^{[s]}}{\Psi_1^{[s]}+\Psi_2^{[s]}+\Psi_3^{[s]}+\epsilon} \\ 1 \end{bmatrix} \quad \underline{\Phi}_2(k) = \begin{bmatrix} \frac{\Psi_2^{[s]}}{\Psi_1^{[s]}+\Psi_2^{[s]}+\Psi_3^{[s]}+\epsilon} \\ \frac{\Psi_2^{[s]}}{\Psi_1^{[s]}+\Psi_2^{[s]}+\Psi_3^{[s]}+\epsilon} \\ 0 \end{bmatrix} \quad \underline{\Phi}_3(k) = \begin{bmatrix} \frac{\Psi_3^{[s]}}{\Psi_1^{[s]}+\Psi_2^{[s]}+\Psi_3^{[s]}+\epsilon} \\ \frac{\Psi_3^{[s]}}{\Psi_1^{[s]}+\Psi_2^{[s]}+\Psi_3^{[s]}+\epsilon} \\ 0 \end{bmatrix} . \tag{B.7}$$

Note that the first two rows of $\underline{\Phi}_1(k)$, $\underline{\Phi}_2(k)$, and $\underline{\Phi}_3(k)$ contain the same values, respectively, as the validity function is shared for the MIMO LMN. The argument k is omitted for brevity.

Normalized Radial Basis Functions – Example 2 As another example, consider a second-order LMSSN with a MISO LMN with two LMs for the first state equation, and a MISO LMN with three LMs for the second state equation, and an affine output equation. Then, (B.6) becomes

$$\underline{\Phi}_1(k) = \begin{bmatrix} \frac{\Psi_{1,1}^{[s]}}{\Psi_{1,1}^{[s]}+\Psi_{1,2}^{[s]}+\epsilon} \\ \frac{\Psi_{2,1}^{[s]}}{\Psi_{2,1}^{[s]}+\Psi_{2,2}^{[s]}+\Psi_{2,3}^{[s]}+\epsilon} \\ 1 \end{bmatrix} \quad \underline{\Phi}_2(k) = \begin{bmatrix} \frac{\Psi_{1,2}^{[s]}}{\Psi_{1,1}^{[s]}+\Psi_{1,2}^{[s]}+\epsilon} \\ \frac{\Psi_{2,2}^{[s]}}{\Psi_{2,1}^{[s]}+\Psi_{2,2}^{[s]}+\Psi_{2,3}^{[s]}+\epsilon} \\ 0 \end{bmatrix} \quad \underline{\Phi}_3(k) = \begin{bmatrix} 0 \\ \frac{\Psi_{2,3}^{[s]}}{\Psi_{2,1}^{[s]}+\Psi_{2,2}^{[s]}+\Psi_{2,3}^{[s]}+\epsilon} \\ 0 \end{bmatrix} . \tag{B.8}$$

Here, the argument k is omitted as well for brevity. Note that in $\Psi_{i,j}^{[s]}$, the first index indicates the i-th state equation and the second index stands for the j-th local model.

Radial Basis Functions The RBFs are calculated by

$$\underline{\Psi}_j(k) = \exp\left(-\frac{1}{2}\left(\left[\underline{\hat{x}}^T(k) \quad \underline{u}^T(k)\right] - \underline{\mu}_j^T\right) \cdot \underline{\Sigma}_j^{-1} \cdot \left(\left[\underline{\hat{x}}^T(k) \quad \underline{u}^T(k)\right]^T - \underline{\mu}_j\right)\right) , \tag{B.9}$$

where $\underline{\mu}_j^T$ (row vector) contains the center coordinates of the RBFs regarding the state and input dimensions

$$\underline{\mu}_j = \begin{bmatrix} \underline{\mu}_j^{[x]} \\ \underline{\mu}_j^{[u]} \end{bmatrix} , \tag{B.10}$$

and $\underline{\Sigma}_j$ is a diagonal matrix, which contains the squared standard deviations regarding the state and input dimensions

$$\underline{\Sigma}_j = \begin{bmatrix} \underline{\Sigma}_j^{[x]} & \underline{0} \\ \underline{0} & \underline{\Sigma}_j^{[u]} \end{bmatrix} . \tag{B.11}$$

Hierarchical Sigmoid Functions The hierarchical sigmoid validity functions are calculated by

$$\underline{\Phi}_j(k) = \prod_{l=1}^{n_c} \underline{\Psi}_l(k) , \tag{B.12}$$

where n_c is the number of stacked sigmoid functions. The sigmoid functions $\underline{\Psi}_l(k)$ are calculated by

$$\underline{\Psi}_l(k) = \frac{1}{1 + e^{-\underline{\zeta}_l(k)}} \quad \text{with} \tag{B.13}$$

$$\underline{\zeta}_l(k) = \kappa \left(\underline{v}_l^T \begin{bmatrix} 1 & \underline{\hat{x}}^T(k) & \underline{u}^T(k) \end{bmatrix}^T \right) . \tag{B.14}$$

Here, κ is a fiddle parameter determining the steepness of the sigmoid functions and $\underline{v}_l^T$ are the split parameters containing the distance to the origin and "direction" of the sigmoid (which can be optimized as well) as

$$\underline{v}_l = \begin{bmatrix} v_{l0} \\ \underline{v}_l^{[x]} \\ \underline{v}_l^{[u]} \end{bmatrix} . \tag{B.15}$$

Optimization Problem All local model parameters are gathered in

$$\underline{\Theta} = \begin{bmatrix} \underline{\Theta}_1 & \underline{\Theta}_2 & \dots & \underline{\Theta}_{\max(n_{m_i}, n_o)} \end{bmatrix} . \tag{B.16}$$

In the most general case, all optimizable parameters are finally concatenated in the vector

$$\underline{\theta} \; = \; [\mathrm{vec}(\underline{\Theta})^T \;\; \underline{V}^T \;\; \underline{x}_0^T]^T . \tag{B.17}$$

Here, vec$(\cdot)$ selects all non-empty elements of $\underline{\Theta}$ and vectorizes them. The vector $\underline{V}$ stores all split parameters $\underline{v}_l$ from (B.14) of all sigmoid functions in the hierarchical local model tree (HILOMOT) case, while $\underline{x}_0^T$ is the initial training state vector. Optimizing the split parameters $\underline{V}$ and initial training state vector $\underline{x}_0^T$ is optional. It shall be noted that the majority of parameters in $\underline{\theta}$ comes from vec$(\underline{\Theta})$, a lot fewer from $\underline{V}$, and only n_x from $\underline{x}_0$.

The optimization problem is then stated by

$$\arg\min_{\underline{\theta}} I(\underline{\theta}) = \arg\min_{\underline{\theta}} \sum_{k=1}^{N} e^2(k, \underline{\theta}) \quad \text{with} \tag{B.18}$$

$$e(k, \underline{\theta}) = y(k) - \hat{y}(k, \underline{\theta})\,. \tag{B.19}$$

B.2 Affine Transformation

Starting point for the affine transformation are the MISO LMSSN equations from (B.1). The state trajectory is transformed initially and after every optimization in such a way that it scales exactly in the unit hypercube $[0,1]^{n_x}$ (see Fig. B.1). To accomplish this, for each state variable $\hat{x}_i$ an offset s_i^o and range parameter s_i^r is calculated according to

$$s_i^o = \min_k \hat{x}_i(k)\,, \tag{B.20}$$

$$s_i^r = \max_k \hat{x}_i(k) - \min_k \hat{x}_i(k)\,. \tag{B.21}$$

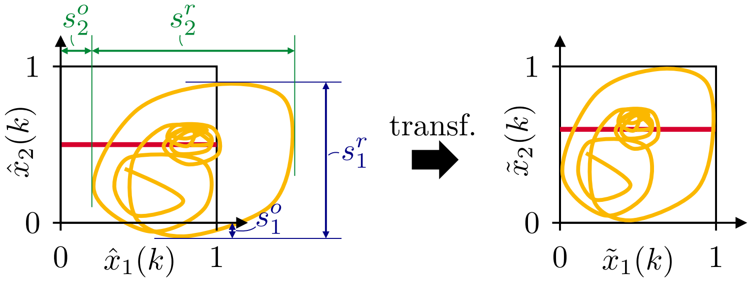


Figure B.1: Transformation of state trajectory for a two-dimensional example. The evolving state trajectory is transformed so that it fits inside the square $[0,1]^2$.

Now, the transformed state vector $\underline{\tilde{x}}(k)$ can be found by

$$\underline{\tilde{x}}(k) = \underline{T}^{-1}(\underline{\hat{x}}(k) - \underline{t}) , \tag{B.22}$$

where

$$\underline{T} = \begin{bmatrix} s_1^r & 0 & \cdots & 0 \\ 0 & s_2^r & & \\ \vdots & & \ddots & \vdots \\ 0 & & \cdots & s_{n_x}^r \end{bmatrix} \quad \text{and} \quad \underline{t} = \begin{bmatrix} s_1^o \\ s_2^o \\ \vdots \\ s_{n_x}^o \end{bmatrix} . \tag{B.23}$$

Using the relation (B.22) to transform (B.1) leads to

$$\begin{aligned} \underline{\tilde{x}}(k+1) &= \sum_{j=1}^{\max(n_{m_i},n_o)} \left(\underline{\tilde{o}}_j + \underline{\tilde{A}}_j \underline{\tilde{x}}(k) + \underline{\tilde{B}}_j \underline{u}(k) \right) \Phi_j^{[s]} \left(\underline{T}\, \underline{\tilde{x}}(k) + \underline{t}, u(k) \right) \\ \hat{y}(k) &= \sum_{j=1}^{\max(n_{m_i},n_o)} \left(\tilde{p}_j + \underline{\tilde{c}}_j^T \underline{\tilde{x}}(k) + \underline{\tilde{d}}_j^T \underline{u}(k) \right) \Phi_j^{[o]} \left(\underline{T}\, \underline{\tilde{x}}(k) + \underline{t}, u(k) \right) , \end{aligned} \tag{B.24}$$

with the transformed scalars, vectors, and matrices

$$\begin{aligned} \underline{\tilde{o}}_j &= \underline{T}^{-1} \underline{o}_j + \underline{T}^{-1} \underline{A}_j \underline{T}\, \underline{t} - \underline{t} \\ \underline{\tilde{A}}_j &= \underline{T}^{-1} \underline{A}_j \underline{T} \\ \underline{\tilde{B}}_j &= \underline{T}^{-1} \underline{B}_j \\ \tilde{p}_j &= p_j + \underline{c}_j^T \underline{T}\, \underline{t} \\ \underline{\tilde{c}}_j^T &= \underline{c}_j^T \underline{T} \\ \underline{\tilde{d}}_j^T &= \underline{d}_j^T . \end{aligned} \tag{B.25}$$

Transforming NRBF Validity Functions The transformation of the NRBF validity functions leads to transformed center coordinates for the state variables (compare (B.10))

$$\underline{\tilde{\mu}}^{[x]^T} = (\underline{\mu}^{[x]^T} - \underline{t}^T)\underline{T}^{-1} \tag{B.26}$$

and transformed squared standard deviations (compare (B.11)) for the state variables

$$\underline{\tilde{\Sigma}}^{[x]^{-1}} = \underline{T}^T \underline{\Sigma}^{[x]^{-1}} \underline{T} . \tag{B.27}$$

Since both $\underline{T}$ and $\underline{\Sigma}^{[x]}$ are diagonal matrices, (B.27) simplifies to

$$\underline{\tilde{\Sigma}}^{[x]} = (\underline{T}^{-1})^2 \underline{\Sigma}^{[x]} . \tag{B.28}$$

The center coordinate in the process input dimensions $\underline{\mu}^{[u]}$ and the squared standard deviation regarding the process input dimensions $\underline{\Sigma}^{[u]}$ stay unaltered.

Transforming Hierarchical Sigmoid Validity Functions For every sigmoid function, the v_{l0} and all entries of $\underline{v}_l^{[x]}$ (compare (B.15)) are transformed as

$$\tilde{v}_{l0} = v_{l0} + \underline{v}_l^{[x]^T} \underline{t} \tag{B.29}$$

$$\underline{\tilde{v}}_l^{[x]^T} = \underline{v}_l^{[x]^T} \underline{T} \,. \tag{B.30}$$

The entries in $\underline{v}_l^{[u]}$ regarding the process input dimensions stay unaltered.

Initial Condition The initial condition for the state vector is transformed as

$$\underline{\tilde{x}}_0 = \underline{T}^{-1}(\underline{\hat{x}}_0 - \underline{t}) \,. \tag{B.31}$$

B.3 Gradient Calculations

To better understand the derivation of the gradients, we will first consider a more intuitive and schematic sketch that is not so strict regarding notation. Then, in a second step, we will derive the exact gradients.

B.3.1 Sketch of Gradient Calculation

We consider the MISO state space model

$$\underline{\hat{x}}(k+1) = \underline{o}\,\underline{\Phi}(k) + \underline{A}\,\underline{\Phi}(k)\underline{\hat{x}}(k) + \underline{B}\,\underline{\Phi}(k)\underline{u}(k) \tag{B.32}$$

$$\hat{y}(k) = p\,\underline{\Phi}(k) + \underline{c}^T\,\underline{\Phi}(k)\underline{\hat{x}}(k) + \underline{d}^T\,\underline{\Phi}(k)\underline{u}(k) \,. \tag{B.33}$$

The multiplication operation between the parameters ($\underline{o}$, $\underline{A}$, $\underline{B}$, p, $\underline{c}^T$, $\underline{d}^T$), validity vector $\underline{\Phi}(k)$, and state vector $\underline{\hat{x}}(k)$ or input $\underline{u}(k)$ is not closely defined. All model parameters are stored in the matrix

$$\underline{\Theta} = \begin{bmatrix} \underline{o} & \underline{A} & \underline{B} \\ p & \underline{c}^T & \underline{d}^T \end{bmatrix} \,. \tag{B.34}$$

All optimizable parameters are concatenated in the vector

$$\underline{\theta} \;=\; [\mathrm{vec}(\underline{\Theta})^T \;\; \underline{V}^T \;\; \underline{x}_0^T]^T \,. \tag{B.35}$$

Here, vec$(\cdot)$ selects all non-empty elements of $\underline{\Theta}$ and vectorizes them. The vector $\underline{V}$ stores all split parameters $\underline{v}_l$ from (B.14) of all sigmoid functions in the HILOMOT case, while $\underline{x}_0^T$ is the initial training state vector.

Calculating the derivative of the model output $\hat{y}(k)$ with respect to θ_i (i-th entry of $\underline{\theta}$) yields

$$\begin{aligned}\frac{\partial \hat{y}(k)}{\partial \theta_i} =& p\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} + \frac{\partial p}{\partial \theta_i}\underline{\Phi}(k) \\ &+ \left(\underline{c}^T\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} + \frac{\partial \underline{c}^T}{\partial \theta_i}\underline{\Phi}(k)\right)\underline{\hat{x}}(k) + \underline{c}^T\underline{\Phi}(k)\frac{\partial \underline{\hat{x}}(k)}{\partial \theta_i} \\ &+ \left(\underline{d}^T\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} + \frac{\partial \underline{d}^T}{\partial \theta_i}\underline{\Phi}(k)\right)\underline{u}(k)\end{aligned} \tag{B.36}$$

with

$$\begin{aligned}\frac{\partial \underline{\hat{x}}(k+1)}{\partial \theta_i} =& \underline{o}\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} + \frac{\partial \underline{o}}{\partial \theta_i}\underline{\Phi}(k) \\ &+ \left(\underline{A}\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} + \frac{\partial \underline{A}}{\partial \theta_i}\underline{\Phi}(k)\right)\underline{\hat{x}}(k) + \underline{A}\,\underline{\Phi}(k)\frac{\partial \underline{\hat{x}}(k)}{\partial \theta_i} \\ &+ \left(\underline{B}\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} + \frac{\partial \underline{B}}{\partial \theta_i}\underline{\Phi}(k)\right)\underline{u}(k)\,.\end{aligned} \tag{B.37}$$

Normalized Radial Basis Function Validity Function Gradient In the case of NRBFs as validity functions

$$\underline{\Phi}(k) = \frac{\underline{\Psi}(k)}{\left\|\underline{\Psi}(k)\right\|_1}\,, \tag{B.38}$$

the derivatives are calculated as

$$\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} = \frac{\frac{\partial \underline{\Psi}(k)}{\partial \theta_i}\left\|\underline{\Psi}(k)\right\|_1 - \underline{\Psi}(k)\left\|\frac{\partial \underline{\Psi}(k)}{\partial \theta_i}\right\|_1}{\left\|\underline{\Psi}(k)\right\|_1^2} \tag{B.39}$$

with

$$\frac{\partial \underline{\Psi}(k)}{\partial \theta_i} = -\,\underline{\Psi}(k)\left(\left[\underline{\hat{x}}^T(k)\;\;\underline{u}^T(k)\right] - \underline{\mu}^T\right)\underline{\Sigma}^{-1}\left[\frac{\partial \underline{\hat{x}}^T(k)}{\partial \theta_i}\;\;\underline{0}^{1\times n_p}\right]^T\,, \tag{B.40}$$

where $\underline{\mu}$ contains the center coordinates of the RBFs regarding the state and input dimensions, and $\underline{\Sigma}$ is a diagonal matrix, containing the squared standard deviations regarding the state and process input dimensions.

Hierarchical Sigmoid Validity Function Gradient In the case of hierarchical sigmoid validity functions

$$\underline{\Phi}(k) = \prod_{l=1}^{n_c} \underline{\Psi}_l(k) \tag{B.41}$$

$$\underline{\Psi}_l(k) = \frac{1}{1 + e^{-\underline{\zeta}_l(k)}} \tag{B.42}$$

$$\underline{\zeta}_l(k) = \kappa \left(\underline{v}_l^T \begin{bmatrix} 1 & \underline{\hat{x}}^T(k) & \underline{u}^T(k) \end{bmatrix}^T \right) , \tag{B.43}$$

where κ is a fiddle parameter determining the steepness of the sigmoid functions. The split parameters $\underline{v}_l^T$ containing the "direction" of the sigmoid can be optimized as well. The derivatives are calculated as

$$\frac{\partial \underline{\Phi}(k)}{\partial \theta_i} = \sum_{l=1}^{n_c} \frac{\partial \underline{\Psi}_l(k)}{\partial \theta_i} \prod_{\substack{j=1 \\ j \neq l}}^{n_c} \underline{\Psi}_j(k) \tag{B.44}$$

$$\frac{\partial \underline{\Psi}_l(k)}{\partial \theta_i} = \underline{\Psi}_l(k) \frac{1}{1 + e^{\underline{\zeta}_l(k)}} \frac{\partial \underline{\zeta}_l(k)}{\partial \theta_i} \tag{B.45}$$

$$\frac{\partial \underline{\zeta}_l(k)}{\partial \theta_i} = \kappa \left(\frac{\underline{v}_l^T}{\partial \theta_i} \begin{bmatrix} 1 & \underline{\hat{x}}^T(k) & \underline{u}^T(k) \end{bmatrix}^T + \underline{v}_l^T \begin{bmatrix} 0 & \frac{\partial \underline{\hat{x}}^T(k)}{\partial \theta_i} & \underline{0}^{1 \times n_p} \end{bmatrix}^T \right) , \tag{B.46}$$

where n_c is the number of hierarchically stacked sigmoid functions.

B.3.2 Detailed Gradient Calculation

Now, let us consider the general MIMO LMSSN and be more precise about notation. We will start with some definitions. Each parameter matrix consists of multiple "slices", where each row in each "slice" determines the local behavior per state or output equation. The first "slice" in all parameter matrices is fully populated since we always start with an affine approximation. In all other "slices", some rows may contain parameters, while others do not, depending on the way each individual state or output equation is split. Therefore, n_m here corresponds to the maximum number of splits over all equations and all missing parameter entries are filled with zeros. The same goes for the validity function vectors and later for the way the sigmoid validity functions are built for up to n_c stacked sigmoid functions. For later used element-wise divisions, a very small ϵ is added to the denominator.

Now, we define

$$\begin{aligned}
\underline{O}^* &= [\underline{o}_1 \ \ \underline{o}_2 \ \ \dots \ \ \underline{o}_{n_m}] \in \mathbb{R}^{n_x \times n_m} \\
\underline{A}^* &= [\underline{A}_1 \ \ \underline{A}_2 \ \ \dots \ \ \underline{A}_{n_m}] \in \mathbb{R}^{n_x \times n_x n_m} \\
\underline{B}^* &= [\underline{B}_1 \ \ \underline{B}_2 \ \ \dots \ \ \underline{B}_{n_m}] \in \mathbb{R}^{n_x \times n_p n_m} \\
\underline{P}^* &= \left[\underline{p}_1 \ \ \underline{p}_2 \ \ \dots \ \ \underline{p}_{n_m}\right] \in \mathbb{R}^{n_q \times n_m} \\
\underline{C}^* &= [\underline{C}_1 \ \ \underline{C}_2 \ \ \dots \ \ \underline{C}_{n_m}] \in \mathbb{R}^{n_q \times n_x n_m} \\
\underline{D}^* &= [\underline{D}_1 \ \ \underline{D}_2 \ \ \dots \ \ \underline{D}_{n_m}] \in \mathbb{R}^{n_q \times n_p n_m} \,,
\end{aligned} \tag{B.47}$$

where all “parameter slices” are concatenated over the second dimension. The full parameter matrix is defined as

$$\underline{\Theta}^* = \begin{bmatrix} \underline{O}^* & \underline{A}^* & \underline{B}^* \\ \underline{P}^* & \underline{C}^* & \underline{D}^* \end{bmatrix} . \tag{B.48}$$

All optimizable parameters are concatenated in the vector

$$\underline{\theta} \ = \ [\mathrm{vec}(\underline{\Theta}^*)^T \ \ \underline{V}^T \ \ \underline{x}_0^T]^T \,, \tag{B.49}$$

where vec(·) selects all non-empty elements of $\underline{\Theta}^*$ and vectorizes them. Parameter θ_i is the i-th element of vector $\underline{\theta}$.

The validity function vectors are also concatenated in the same manner as

$$\underline{\Phi}^{*[s]}(k) = \left[\underline{\Phi}_1^{[s]}(k) \ \ \underline{\Phi}_2^{[s]}(k) \ \ \dots \ \ \underline{\Phi}_{n_m}^{[s]}(k)\right] \in \mathbb{R}^{n_x \times n_m} \tag{B.50}$$

$$\underline{\Phi}^{*[o]}(k) = \left[\underline{\Phi}_1^{[o]}(k) \ \ \underline{\Phi}_2^{[o]}(k) \ \ \dots \ \ \underline{\Phi}_{n_m}^{[o]}(k)\right] \in \mathbb{R}^{n_q \times n_m} \tag{B.51}$$

and the membership function matrices $\underline{\Psi}^{*[s]}(k)$ and $\underline{\Psi}^{*[o]}(k)$ as well. The L_1-norms $\left\|\underline{\Psi}^{*[s]}(k)\right\|_1$ and $\left\|\underline{\Psi}^{*[o]}(k)\right\|_1$ are standing vectors with row-wise L_1-norms. Since the membership function values are always positive, this is equivalent to a sum over the columns over the matrix.

The operator $\odot$ denotes the Hadamard (element-wise) product, the operator $\oslash$ denotes the Hadamard (element-wise) division (see Appx. B.3.3), and $\otimes$ denotes the Kronecker product (see Appx. B.3.4).

The last definitions we need to get started are

$$\underline{1}^{a\times b} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} , \tag{B.52}$$

which is a matrix of size $(a \times b)$ with all entries being one and

$$\underline{0}^{a\times b} \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \tag{B.53}$$

is a matrix of size $(a \times b)$ with all entries being zero.

LMSSN Equations We will write the LMSSN with the introduced notation as

$$\begin{aligned} \underline{\hat{x}}(k+1) &= \left(\underline{O}^{*} \odot \underline{\Phi}^{*[s]}\right) \underline{1}^{n_m\times 1} \\ &\quad + \left[\underline{A}^{*} \odot \left(\underline{\Phi}^{*[s]} \otimes \underline{1}^{1\times n_x}\right)\right] \left(\underline{1}^{n_m\times 1} \otimes \underline{\hat{x}}(k)\right) \\ &\quad + \left[\underline{B}^{*} \odot \left(\underline{\Phi}^{*[s]} \otimes \underline{1}^{1\times n_p}\right)\right] \left(\underline{1}^{n_m\times 1} \otimes \underline{u}(k)\right) \end{aligned} \tag{B.54}$$

$$\begin{aligned} \underline{\hat{y}}(k) &= \left(\underline{P}^{*} \odot \underline{\Phi}^{*[o]}\right) \underline{1}^{n_m\times 1} \\ &\quad + \left[\underline{C}^{*} \odot \left(\underline{\Phi}^{*[o]} \otimes \underline{1}^{1\times n_x}\right)\right] \left(\underline{1}^{n_m\times 1} \otimes \underline{\hat{x}}(k)\right) \\ &\quad + \left[\underline{D}^{*} \odot \left(\underline{\Phi}^{*[o]} \otimes \underline{1}^{1\times n_p}\right)\right] \left(\underline{1}^{n_m\times 1} \otimes \underline{u}(k)\right) . \end{aligned} \tag{B.55}$$

State Equation Term – Example 1 As an example, we take the second row of (B.54) for a second-order LMSSN with a MIMO LMN with three LMs in the state equation. Then, the second row of (B.54) becomes

$$\left[\left[\begin{array}{cc|cc|cc} a_{(1,1),1} & a_{(1,2),1} & a_{(1,1),2} & a_{(1,2),2} & a_{(1,1),3} & a_{(1,2),3} \\ a_{(2,1),1} & a_{(2,2),1} & a_{(2,1),2} & a_{(2,2),2} & a_{(2,1),3} & a_{(2,2),3} \end{array}\right] \odot \left(\left[\begin{array}{c|c|c} \Phi_1^{[s]} & \Phi_2^{[s]} & \Phi_3^{[s]} \\ \Phi_1^{[s]} & \Phi_2^{[s]} & \Phi_3^{[s]} \end{array}\right] \otimes \begin{bmatrix} 1 & 1 \end{bmatrix}\right)\right] \\ \cdot \left(\left[\begin{array}{c} 1 \\ \hline 1 \\ \hline 1 \end{array}\right] \otimes \begin{bmatrix} \hat{x}_1(k) \\ \hat{x}_2(k) \end{bmatrix}\right)$$

$$= \left[\begin{array}{cc|cc|cc} a_{(1,1),1}\mathit{\Phi}_1^{[s]} & a_{(1,2),1}\mathit{\Phi}_1^{[s]} & a_{(1,1),2}\mathit{\Phi}_2^{[s]} & a_{(1,2),2}\mathit{\Phi}_2^{[s]} & a_{(1,1),3}\mathit{\Phi}_3^{[s]} & a_{(1,2),3}\mathit{\Phi}_3^{[s]} \\ a_{(2,1),1}\mathit{\Phi}_1^{[s]} & a_{(2,2),1}\mathit{\Phi}_1^{[s]} & a_{(2,1),2}\mathit{\Phi}_2^{[s]} & a_{(2,2),2}\mathit{\Phi}_2^{[s]} & a_{(2,1),3}\mathit{\Phi}_3^{[s]} & a_{(2,2),3}\mathit{\Phi}_3^{[s]} \end{array}\right] \left[\begin{array}{c} \hat{x}_1(k) \\ \hat{x}_2(k) \\ \hline \hat{x}_1(k) \\ \hat{x}_2(k) \\ \hline \hat{x}_1(k) \\ \hat{x}_2(k) \end{array}\right] . \tag{B.56}$$

The parameters $a_{(i,l),j}$ indicate the LM parameter for the i-th state equation, which is multiplied by the l-th state in the j-th local model. The lines within the matrices are drawn between different LMs.

State Equation Term – Example 2 As another example, consider a second-order LMSSN with a MISO LMN with two LMs for the first state equation and a MISO LMN with three LMs for the second state equation. Then, the second row of (B.54) becomes

$$\left[\left[\begin{array}{cc|cc|cc} a_{(1,1),1} & a_{(1,2),1} & a_{(1,1),2} & a_{(1,2),2} & 0 & 0 \\ a_{(2,1),1} & a_{(2,2),1} & a_{(2,1),2} & a_{(2,2),2} & a_{(2,1),3} & a_{(2,2),3} \end{array}\right] \odot \left(\left[\begin{array}{c|c|c} \mathit{\Phi}_{1,1}^{[s]} & \mathit{\Phi}_{1,2}^{[s]} & 0 \\ \mathit{\Phi}_{2,1}^{[s]} & \mathit{\Phi}_{2,2}^{[s]} & \mathit{\Phi}_{2,3}^{[s]} \end{array}\right] \otimes \begin{bmatrix} 1 & 1 \end{bmatrix}\right)\right] \cdot \left(\left[\begin{array}{c} 1 \\ \hline 1 \\ \hline 1 \end{array}\right] \otimes \begin{bmatrix} \hat{x}_1(k) \\ \hat{x}_2(k) \end{bmatrix}\right)$$

$$= \left[\begin{array}{cc|cc|cc} a_{(1,1),1}\mathit{\Phi}_{1,1}^{[s]} & a_{(1,2),1}\mathit{\Phi}_{1,1}^{[s]} & a_{(1,1),2}\mathit{\Phi}_{1,2}^{[s]} & a_{(1,2),2}\mathit{\Phi}_{1,2}^{[s]} & 0 & 0 \\ a_{(2,1),1}\mathit{\Phi}_{2,1}^{[s]} & a_{(2,2),1}\mathit{\Phi}_{2,1}^{[s]} & a_{(2,1),2}\mathit{\Phi}_{2,2}^{[s]} & a_{(2,2),2}\mathit{\Phi}_{2,2}^{[s]} & a_{(2,1),3}\mathit{\Phi}_{2,3}^{[s]} & a_{(2,2),3}\mathit{\Phi}_{2,3}^{[s]} \end{array}\right] \left[\begin{array}{c} \hat{x}_1(k) \\ \hat{x}_2(k) \\ \hline \hat{x}_1(k) \\ \hat{x}_2(k) \\ \hline \hat{x}_1(k) \\ \hat{x}_2(k) \end{array}\right] . \tag{B.57}$$

The parameters $a_{(i,l),j}$ indicate the LM parameter for the i-th state equation, which is multiplied by the l-th state in the j-th local model. The lines within the matrices are drawn between different LMs.

Gradient Calculation The gradient calculation yields for the output equations

$$\begin{aligned}\frac{\partial \underline{\hat{y}}(k)}{\partial \theta_i} = {} & \left(\underline{P}^* \odot \frac{\partial \underline{\Phi}^{*[o]}}{\partial \theta_i}\right) \underline{1}^{n_m \times 1} + \left(\frac{\partial \underline{P}^*}{\partial \theta_i} \odot \underline{\Phi}^{*[o]}\right) \underline{1}^{n_m \times 1} \\ & + \left[\underline{C}^* \odot \left(\frac{\partial \underline{\Phi}^{*[o]}}{\partial \theta_i} \otimes \underline{1}^{1 \times n_x}\right) + \frac{\partial \underline{C}^*}{\partial \theta_i} \odot \left(\underline{\Phi}^{*[o]} \otimes \underline{1}^{1 \times n_x}\right)\right] \left(\underline{1}^{n_m \times 1} \otimes \underline{\hat{x}}(k)\right) \\ & + \left[\underline{C}^* \odot \left(\underline{\Phi}^{*[o]} \otimes \underline{1}^{1 \times n_x}\right)\right] \left(\underline{1}^{n_m \times 1} \otimes \frac{\partial \underline{\hat{x}}(k)}{\partial \theta_i}\right) \\ & + \left[\underline{D}^* \odot \left(\frac{\partial \underline{\Phi}^{*[o]}}{\partial \theta_i} \otimes \underline{1}^{1 \times n_p}\right) + \frac{\partial \underline{D}^*}{\partial \theta_i} \odot \left(\underline{\Phi}^{*[o]} \otimes \underline{1}^{1 \times n_p}\right)\right] \left(\underline{1}^{n_m \times 1} \otimes \underline{u}(k)\right)\end{aligned} \tag{B.58}$$

and for the state equations

$$\begin{aligned}\frac{\partial \underline{\hat{x}}(k+1)}{\partial \theta_i} = {} & \left(\underline{O}^* \odot \frac{\partial \underline{\Phi}^{*[s]}}{\partial \theta_i}\right) \underline{1}^{n_m \times 1} + \left(\frac{\partial \underline{O}^*}{\partial \theta_i} \odot \underline{\Phi}^{*[s]}\right) \underline{1}^{n_m \times 1} \\ & + \left[\underline{A}^* \odot \left(\frac{\partial \underline{\Phi}^{*[s]}}{\partial \theta_i} \otimes \underline{1}^{1 \times n_x}\right) + \frac{\partial \underline{A}^*}{\partial \theta_i} \odot \left(\underline{\Phi}^{*[s]} \otimes \underline{1}^{1 \times n_x}\right)\right] \left(\underline{1}^{n_m \times 1} \otimes \underline{\hat{x}}(k)\right) \\ & + \left[\underline{A}^* \odot \left(\underline{\Phi}^{*[s]} \otimes \underline{1}^{1 \times n_x}\right)\right] \left(\underline{1}^{n_m \times 1} \otimes \frac{\partial \underline{\hat{x}}(k)}{\partial \theta_i}\right) \\ & + \left[\underline{B}^* \odot \left(\frac{\partial \underline{\Phi}^{*[s]}}{\partial \theta_i} \otimes \underline{1}^{1 \times n_p}\right) + \frac{\partial \underline{B}^*}{\partial \theta_i} \odot \left(\underline{\Phi}^{*[s]} \otimes \underline{1}^{1 \times n_p}\right)\right] \left(\underline{1}^{n_m \times 1} \otimes \underline{u}(k)\right) .\end{aligned} \tag{B.59}$$

NRBF Validity Function Gradients The NRBF validity functions are calculated as

$$\underline{\Phi}^{*[s/o]} = \underline{\Psi}^{*[s/o]} \oslash \left(\left\|\underline{\Psi}^{*[s/o]}\right\|_1 \otimes \underline{1}^{1 \times n_m}\right) . \tag{B.60}$$

Note that $(\cdot)^{[s/o]}$ indicates that the calculations can be done for the state and output equations, respectively. The definition of $\underline{\Psi}^{*[s/o]}$ will be done per “slice” $\underline{\Psi}_j^{[s/o]}$ for $j = 1, \ldots, n_m$ since tensor operations are necessary to compute the whole $\underline{\Psi}^{*[s/o]}$ in one step. Therefore, $\underline{\Psi}_j^{[s/o]}$ is calculated as

$$\underline{\Psi}_j^{[s/o]} = \exp\!\left(-\frac{1}{2}\left[\left(\underline{1}^{(n_x/n_q) \times 1} \otimes \left[\underline{\hat{x}}^T(k) \;\; \underline{u}^T(k)\right] - \underline{\mu}_j^{[s/o]^T}\right) \oslash \underline{\sigma}_j^{[s/o]^T}\right]^2 \underline{1}^{n_x + n_p \times 1}\right) , \tag{B.61}$$

where the square operation $(\cdot)^2$ is done element-wise. The center coordinates are stored per "slice" in the matrices $\underline{\mu}_j^{[s/o]^T}$. Those contain in each row the center coordinates regarding all state dimensions and input dimensions, while each column contains the center coordinates regarding one dimension for all state or output equations, respectively. Those matrices are likewise stacked along the second dimension as

$$\underline{\mu}^{*[s/o]^T} = \begin{bmatrix} \underline{\mu}_1^{[s/o]^T} & \underline{\mu}_2^{[s/o]^T} & \dots & \underline{\mu}_{n_m}^{[s/o]^T} \end{bmatrix} \in \mathbb{R}^{(n_x/n_q)\times(n_x+n_p)n_m} \,. \tag{B.62}$$

The standard deviations $\underline{\sigma}^{*[s/o]^T}$ are organized in the same way. Now, the gradients are calculated as

$$\begin{aligned} \frac{\partial \underline{\Phi}^{*[s/o]}}{\partial \theta_i} = \Bigg(& \frac{\partial \underline{\Psi}^{*[s/o]}(k)}{\partial \theta_i} \odot \left(\left\| \underline{\Psi}^{*[s/o]}(k) \right\|_1 \otimes \underline{1}^{1\times n_m} \right) \\ & - \underline{\Psi}^{*[s/o]}(k) \odot \left(\left\| \frac{\partial \underline{\Psi}^{*[s/o]}(k)}{\partial \theta_i} \right\|_1 \otimes \underline{1}^{1\times n_m} \right) \Bigg) \\ & \oslash \left\| \underline{\Psi}^{*[s/o]}(k) \right\|_1^2 \end{aligned} \tag{B.63}$$

with

$$\begin{aligned} \frac{\partial \underline{\Psi}_j^{[s/o]}}{\partial \theta_i} = -\underline{\Psi}_j^{[s/o]} \odot \Bigg(\Bigg[& \left(\underline{1}^{(n_x/n_q)\times 1} \otimes \begin{bmatrix} \underline{\hat{x}}^T(k) & \underline{u}^T(k) \end{bmatrix} - \underline{\mu}_j^{[s/o]^T} \right) \\ & \odot \left(\underline{1}^{(n_x/n_q)\times 1} \otimes \begin{bmatrix} \frac{\partial \underline{\hat{x}}^T(k)}{\partial \theta_i} & \underline{0}^{1\times n_p} \end{bmatrix} \right) \oslash \underline{\sigma}_j^{[s/o]^{T^2}} \Bigg] \underline{1}^{n_x+n_p\times 1} \Bigg) \,. \end{aligned} \tag{B.64}$$

Hierarchical Sigmoid Validity Function Gradients The validity functions $\underline{\Phi}^{*[s/o]}(k)$ consist of element-wise products of the sigmoid functions stored in matrices $\underline{\Psi}_l^{*[s/o]}(k) \in \mathbb{R}^{(n_x/n_q)\times n_m}$. Note that $l = 1, \dots, n_c$ is the index of multiplied sigmoid functions (so to say the depth of the hierarchy) and not the index for the LM. The number n_c may change depending on the state or output equation. All empty entries in $\underline{\Psi}_l^{*[s/o]}(k)$ are filled with zeros if no validity function is present at that location and filled with ones if the number of stacked sigmoids is smaller than n_c in this location. This then yields

$$\underline{\Phi}^{*[s/o]}(k) = \prod_{l=1}^{n_c} \underline{\Psi}_l^{*[s/o]}(k) \,, \tag{B.65}$$

with the product $\prod$ being employed element-wise.

As an example, consider a second-order LMSSN with a MISO LMN with two LMs for the first state equation and a MISO LMN with three LMs for the second state equation. The depth of the hierarchy in the first state equation is one, while it is two for the second state equation. Then, (B.65) becomes

$$\underline{\Phi}^{*[s]}(k) = \left[\begin{array}{c|c|c} \Phi_{1,1}^{[s]} & \Phi_{1,2}^{[s]} & 0 \\ \Phi_{2,1}^{[s]} & \Phi_{2,2}^{[s]} & \Phi_{2,3}^{[s]} \end{array}\right] = \underbrace{\left[\begin{array}{c|c|c} \Psi_{1,1}^{[s]} & 1-\Psi_{1,1}^{[s]} & 0 \\ \Psi_{2,1}^{[s]} & 1-\Psi_{2,1}^{[s]} & 1-\Psi_{2,1}^{[s]} \end{array}\right]}_{\underline{\Psi}_1^{*[s]}} \odot \underbrace{\left[\begin{array}{c|c|c} 1 & 1 & 0 \\ 1 & \Psi_{2,2}^{[s]} & 1-\Psi_{2,2}^{[s]} \end{array}\right]}_{\underline{\Psi}_2^{*[s]}} . \tag{B.66}$$

The sigmoid functions are calculated as

$$\underline{\Psi}_l^{*[s/o]}(k) = \underline{1}^{(n_x/n_q)\times n_m} \oslash \left[\underline{1}^{(n_x/n_q)\times n_m} + \exp\left(-\underline{\zeta}_l^{*[s/o]}(k)\right)\right] , \tag{B.67}$$

with the exponential function being employed element-wise. The $\underline{\zeta}_l^{*[s/o]}(k)$ is the concatenation over all active LMs $\underline{\zeta}_{l,j}^{*[s/o]}(k)$

$$\underline{\zeta}_l^{*[s/o]}(k) = \left[\underline{\zeta}_{l,1}^{[s/o]}(k) \;\; \underline{\zeta}_{l,2}^{[s/o]}(k) \;\; \dots \;\; \underline{\zeta}_{l,n_m}^{[s/o]}(k)\right] \in \mathbb{R}^{(n_x/n_q)\times n_m} . \tag{B.68}$$

Each $\underline{\zeta}_{l,j}^{[s/o]}(k)$ is calculated as

$$\underline{\zeta}_{l,j}^{[s/o]}(k) = \kappa \odot \left[\left(\underline{1}^{(n_x/n_q)\times 1} \otimes \underline{v}_{l,j}^{[s/o]}\right) \left[1 \;\; \underline{\hat{x}}^T(k) \;\; \underline{u}^T(k)\right]^T\right] \tag{B.69}$$

with

$$\underline{v}_{l,j}^{[s/o]} \in \mathbb{R}^{1\times 1+n_x+n_p} , \tag{B.70}$$

containing all split parameters for one LM in one state or output equation. The split parameters $\underline{v}_{l,m}^{[s/o]}$ may be optimized as well.

The gradient calculation then yields

$$\frac{\partial \underline{\Phi}^{*[s/o]}(k)}{\partial \theta_i} = \sum_{m=1}^{n_c} \frac{\partial \underline{\Psi}_m^{*[s/o]}(k)}{\partial \theta_i} \prod_{\substack{l=1 \\ l\neq m}}^{n_c} \underline{\Psi}_l^{*[s/o]}(k) \tag{B.71}$$

$$\frac{\partial \underline{\Psi}_l^{*[s/o]}(k)}{\partial \theta_i} = \underline{\Psi}_l^{*[s/o]}(k) \odot \left(\underline{1}^{(n_x/n_q)\times n_m} \oslash \left[\underline{1}^{(n_x/n_q)\times n_m} + \exp\left(\underline{\zeta}_l^{*[s/o]}(k)\right)\right]\right) \odot \frac{\partial \underline{\zeta}_l^{*[s/o]}(k)}{\partial \theta_i} \tag{B.72}$$

$$\begin{aligned}\frac{\partial \underline{\zeta}_{l,m}^{[s/o]}(k)}{\partial \theta_i} = \kappa \odot \Bigg[& \left(\underline{1}^{(n_x/n_q)\times 1} \otimes \frac{\partial \underline{v}_{l,m}^{[s/o]}}{\partial \theta_i}\right) \begin{bmatrix} 1 & \underline{\hat{x}}^T(k) & \underline{u}^T(k) \end{bmatrix}^T + \\ & \left(\underline{1}^{(n_x/n_q)\times 1} \otimes \underline{v}_{l,m}^{[s/o]}\right) \begin{bmatrix} 0 & \frac{\partial \underline{\hat{x}}^T(k)}{\partial \theta_i} & \underline{0}^{1\times n_p} \end{bmatrix}^T \Bigg] .\end{aligned} \tag{B.73}$$

Gradient Calculations with Respect to $\underline{x}_0$ It is common to optimize the initial state $\underline{x}_0$ along with the parameter vector $\underline{\theta}$. This is also usually done during LMSSN optimization. The gradient of the state vector with respect to the i-th initial state $\frac{\partial \underline{\hat{x}}(k)}{\partial \hat{x}_{0,i}}$ is always a zero-vector, except for $k = 0$, where the i-th element is one.

B.3.3 Hadamard Product and Division

The Hadamard product and division, denoted by $\odot$ and $\oslash$, respectively, perform element-wise multiplications or divisions on two equally sized matrices. Let $\underline{A} \in \mathbb{R}^{m \times n}$ and $\underline{B} \in \mathbb{R}^{m \times n}$ be two equally sized matrices. Then the Hadamard product is calculated as

$$\underline{A} \odot \underline{B} = \begin{bmatrix} a_{1,1}b_{1,1} & a_{1,2}b_{1,2} & \cdots & a_{1,n}b_{1,n} \\ a_{2,1}b_{2,1} & a_{2,2}b_{2,2} & & a_{2,n}b_{2,n} \\ \vdots & & \ddots & \vdots \\ a_{m,1}b_{m,1} & a_{m,2}b_{m,2} & \cdots & a_{m,n}b_{m,n} \end{bmatrix}, \tag{B.74}$$

where $a_{i,j}$ and $b_{i,j}$ denote the (i,j)-th entry of the matrices $\underline{A}$ and $\underline{B}$, respectively.

The Hadamard division is defined accordingly as

$$\underline{A} \oslash \underline{B} = \begin{bmatrix} a_{1,1}/b_{1,1} & a_{1,2}/b_{1,2} & \cdots & a_{1,n}/b_{1,n} \\ a_{2,1}/b_{2,1} & a_{2,2}/b_{2,2} & & a_{2,n}/b_{2,n} \\ \vdots & & \ddots & \vdots \\ a_{m,1}/b_{m,1} & a_{m,2}/b_{m,2} & \cdots & a_{m,n}/b_{m,n} \end{bmatrix}. \tag{B.75}$$

B.3.4 Kronecker Product

The Kronecker product, denoted by the operator $\otimes$, can be applied to two arbitrary-sized matrices. Let $\underline{A} \in \mathbb{R}^{m \times n}$ and $\underline{B} \in \mathbb{R}^{o \times p}$ be two arbitrary matrices. The Kronecker product is then calculated as

$$\underline{A} \otimes \underline{B} = \begin{bmatrix} a_{1,1}\underline{B} & a_{1,2}\underline{B} & \cdots & a_{1,n}\underline{B} \\ a_{2,1}\underline{B} & a_{2,2}\underline{B} & & a_{2,n}\underline{B} \\ \vdots & & \ddots & \vdots \\ a_{m,1}\underline{B} & a_{m,2}\underline{B} & \cdots & a_{m,n}\underline{B} \end{bmatrix}, \tag{B.76}$$

where $a_{i,j}$ denotes the (i,j)-th entry of the matrix $\underline{A}$. The resulting matrix has the size $(m \cdot o \times n \cdot p)$.

B.4 LMSSN Resembling Block-oriented Structures

Table B.1 summarizes how the domains and the ranges of the LMSSN state and output equation change when they resemble block-oriented structures (i.e., "nl: $m+1 \to m$" means that a nonlinear function maps from $\mathbb{R}^{m+1} \to \mathbb{R}^m$). The Wiener, Hammerstein, and Hammerstein-Wiener processes are modeled with an LMSSN of order m (light blue background), while the Wiener-Hammerstein with its two linear time-invariant (LTI) blocks is modeled by an LMSSN of order $2m$ (light yellow background), where m is the order of the dynamic blocks. The general nonlinear identification task can be significantly simplified since the dimensionality of the domain and/or range is drastically reduced in all cases. This reduction leads to lower-dimensional functions that have to be approximated and thus reduces the number of parameters that have to be estimated. Note that the LMSSN only comes close to the block-oriented processes and does not match those exactly in some cases.

Table B.1: Reduction of the dimensionality of the domain and range of the LMSSN state and output equation when resembling block-oriented structures. The equations may be nonlinear (nl) or linear mappings (lin). For example, "nl: $m+1 \to m$" means that a nonlinear function maps from $\mathbb{R}^{m+1} \to \mathbb{R}^m$. When the state equation can be modeled with a combination of linear and nonlinear parts, then those are listed below each other in the same cell. The Wiener, Hammerstein, and Hammerstein-Wiener process can be modeled by an LMSSN of order m (🟦), while the Wiener-Hammerstein process has to be modeled by an LMSSN of order $2m$ (🟨).

LMSSN Structure	State equation $\underline{h}(\cdot)$	Output equation $g(\cdot)$
LMSSN order m	nl: $m+1 \to m$	nl: $m+1 \to 1$
Wiener	lin: $m+1 \to m$	nl: $m \to 1$
Hammerstein	nl: $1 \to 1$ lin: $m \to m$	lin: $m \to 1$
Hammerstein-Wiener	nl: $1 \to 1$ lin: $m \to m$	nl: $m \to 1$
LMSSN order $2m$	nl: $2m+1 \to 2m$	nl: $2m+1 \to 1$
Wiener-Hammerstein	nl: $m \to 1$ lin: $m \to m$ lin: $m+1 \to m$	lin: $m \to 1$

References

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

[2] J. Adamy. *Nichtlineare Systeme und Regelungen.* Springer Berlin Heidelberg, 2014.

[3] C. C. Aggarwal. *Neural Networks and Deep Learning.* Springer International Publishing, 2018.

[4] J. M. Alonso, F. Alvarruiz, J. M. Desantes, L. Hernandez, V. Hernandez, and G. Molto. Combining Neural Networks and Genetic Algorithms to Predict and Reduce Diesel Engine Emissions. *IEEE Transactions on Evolutionary Computation*, 11(1):46–55, 2007.

[5] C. Andersson, A. H. Ribeiro, K. Tiels, N. Wahlström, and T. B. Schön. Deep Convolutional Networks in System Identification. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 3670–3676. IEEE, 2019.

[6] A. Bajric. System identification of a linearized hysteretic system using covariance driven stochastic subspace identification. In *Workshop on Nonlinear System Identification Benchmark*, Apr. 2016.

[7] J. Bayer and C. Osendorfer. Learning Stochastic Recurrent Networks. *arXiv preprint arXiv:1411.7610*, 2014.

[8] J. Belz, T. Münker, T. O. Heinz, G. Kampmann, and O. Nelles. Automatic Modeling with Local Model Networks for Benchmark Processes. *IFAC-PapersOnLine*, 50(1):470–475, 2017.

[9] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[10] M. S. Branicky. Introduction to Hybrid Systems. In *Handbook of Networked and Embedded Control Systems*, pages 91–116. Birkhäuser Boston, 2005.

[11] D. S. Broomhead and D. Lowe. Multivariable Functional Interpolation and Adaptive Networks. Technical report, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

[12] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners, 2020.

[13] C. G. Broyden. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, Mar. 1970.

[14] M. Brunot, A. Janot, and F. Carrillo. Continuous-Time Nonlinear Systems Identification with Output Error Method Based on Derivative-Free Optimisation. *IFAC-PapersOnLine*, 50(1):464–469, 2017.

[15] K. P. Burnham and D. R. Anderson. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological Methods & Research*, 33(2):261–304, 2004.

[16] V. Bushaev. ADAM - latest trends in deep learning optimization. Online Article, Oct. 2018.

[17] S. Chen, S. Billings, and P. Grant. Non-linear System Identification Using Neural Networks. *International Journal of Control*, 51(6):1191–1214, 1990.

[18] S. Chen and S. A. Billings. Representations of Non-linear Systems: The NARMAX Model. *International Journal of Control*, 49(3):1013–1032, 1989.

[19] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing 2014*, Oct. 2014.

[20] F. Chollet et al. Keras. https://keras.io, 2015.

[21] Continental Aftermarket. Datasheet UniNOx-Sensor. `https://www.continental-aftermarket.com/media/3749/continental_uninox_salessheet_final.pdf`. accessed on Febuary 25, 2022.

[22] Z. Dai, H. Liu, Q. Le, and M. Tan. CoAtNet: Marrying Convolution and Attention for All Data Sizes. *Advances in Neural Information Processing Systems*, 34, 2021.

[23] J. De Baerdemaeker and W. Saeys. Advanced Control of Combine Harvesters. *IFAC Proceedings Volumes (4th IFAC Conference on Modelling and Control in Agriculture, Horticulture and Post Harvest Industry)*, 46(18):1–5, 2013.

[24] J. de Jesús Rubio. Stable Kalman filter and neural network for the chaotic systems identification. *Journal of the Franklin Institute*, 354(16):7444 – 7462, 2017.

[25] J. Decuyper, J. Schoukens, K. Tiels, and S. Weiland. Decoupling Multivariate Functions Using a Non-Parametric Filtered CPD Approach. *19h IFAC Symposium on System Identification SYSID 2021*, 2021.

[26] J. Decuyper, K. Tiels, M. Runacres, and J. Schoukens. Retrieving highly structured models starting from black-box nonlinear state-space models using polynomial decoupling. *Mechanical Systems and Signal Processing*, 146:106966, 2021.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[28] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[29] M. Enqvist. *Linear Models of Nonlinear Systems.* PhD thesis, Institute of technology, Linköping University, Sweden, 2005.

[30] M. Enqvist and L. Ljung. Linear Approximations of Nonlinear FIR Systems for Separable Input Processes. *Automatica*, 41(3):459–473, 2005.

[31] M. Espinoza, K. Pelckmans, L. Hoegaerts, J. A. Suykens, and B. D. Moor. A Comparative Study of LS-SVM's Applied to the Silver Box Identification Problem. *IFAC Proceedings Volumes*, 37(13):369–374, 2004.

[32] P. Eykhoff. *System Identification Parameter and State Estimation.* Wiley, 1974.

[33] A. Fakhrizadeh Esfahani. *Structure Discrimination and Identification of Nonlinear Systems.* PhD thesis, Vrije Universiteit, Brussel, 2018.

[34] A. Fakhrizadeh Esfahani, P. Dreesen, K. Tiels, J.-P. Noël, and J. Schoukens. Polynomial State-Space Model Decoupling for the Identification of Hysteretic Systems. *IFAC-PapersOnLine*, 50(1):458–463, 2017.

[35] A. Fakhrizadeh Esfahani, P. Dreesen, K. Tiels, J.-P. Noël, and J. Schoukens. Parameter reduction in nonlinear state-space identification of hysteresis. *Mechanical Systems and Signal Processing*, 104:884–895, 2018.

[36] R. Fletcher. A new Approach to Variable Metric Algorithms. *The Computer Journal*, 13(3):317–322, Jan. 1970.

[37] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, Ltd, May 2000.

[38] A. Garulli, S. Paoletti, and A. Vicino. A survey on switched and piecewise affine system identification. *IFAC Proceedings Volumes (16th IFAC Symposium on System Identification)*, 45(16):344–355, 2012.

[39] D. Gedon, N. Wahlström, T. B. Schön, and L. Ljung. Deep State Space Models for Nonlinear System Identification. *19h IFAC Symposium on System Identification SYSID 2021*, 2021.

[40] Z. Ghahramani and S. T. Roweis. Learning Nonlinear Dynamical Systems using an EM Algorithm. In *Advances in neural information processing systems*, pages 431–437, 1999.

[41] F. Giri and E.-W. Bai, editors. *Block-oriented Nonlinear System Identification.* Springer London, 2010.

[42] D. Goldfarb. A Family of Variable-Metric Methods Derived by Variational Means. *Mathematics of Computation*, 24(109):23–26, 1970.

[43] G. H. Golub and C. F. Van Loan. *Matrix Computations*, volume 3. The John Hopkins University Press, 2013.

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

[45] A. Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*, volume 385 of *Studies in Computational Intelligence.* Springer, 2012.

[46] C. Hametner, C. Mayr, and S. Jakubek. Dynamic NOx emission modelling using local model networks. *International Journal of Engine Research*, 15(8):928–933, 2014.

[47] B. Hartmann. *Lokale Modellnetze zur Identifikation und Versuchsplanung nichtlinearer Systeme.* PhD thesis, Universität Siegen, 2013.

[48] B. Hartmann, T. Ebert, T. Fischer, J. Belz, G. Kampmann, and O. Nelles. LMNTOOL — Toolbox zum automatischen Trainieren lokaler Modellnetze. In *Workshop on Computational Intelligence*, 2012.

[49] B. Hartmann and O. Nelles. Adaptive Test Planning for the Calibration of Combustion Engines – Methodology. *Design of Experiments (DoE) in Engine Development*, pages 1–16, Jan. 2013.

[50] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning.* Springer series in statistics. Springer, New York, NY, 2. edition, 2009.

[51] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[52] T. O. Heinz and O. Nelles. Iterative Excitation Signal Design for Nonlinear Dynamic Black-Box Models. *Procedia Computer Science (Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 21st International Conference)*, 112:1054 – 1061, 2017.

[53] H. Hjalmarsson and J. Schoukens. On Direct Identification of Physical Parameters in Non-Linear Models. *IFAC Proceedings Volumes*, 37(13):375–380, 2004.

[54] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[55] S. Hochreiter and J. Schmidhuber. Long Short-term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[56] B. G. Horne and C. L. Giles. An experimental comparison of recurrent neural networks. In *Advances in neural information processing systems*, pages 697–704, 1995.

[57] K. Hornik, M. Stinchcombe, and H. White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[58] T. A. Johansen, R. Shorten, and R. Murray-Smith. On the Interpretation and Identification of Dynamic Takagi-Sugeno Fuzzy Models. *IEEE Transactions on Fuzzy Systems*, 8(3):297–313, 2000.

[59] D. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*, Dec. 2014.

[60] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[61] J. Kocijan, A. Girard, B. Banko, and R. Murray-Smith. Dynamic systems identification with Gaussian processes. *Mathematical and Computer Modelling of Dynamical Systems*, 11(4):411–424, 2005.

[62] T. Kösters, T. O. Heinz, and O. Nelles. Excitation Signal Design – Tackling practical problems with Design of Experiment. In *IFAC Modeling, Estimation and Control Conference MECC 2022*, 2022. Accepted for publication.

[63] R. Kumar, S. Srivastava, J. R. P. Gupta, and A. Mohindru. Comparative study of neural networks for dynamic nonlinear systems identification. *Soft Computing*, 23(1):101–114, Jan. 2019.

[64] L. Lauwers. *Some practical applications of the best linear approximation in nonlinear block-oriented modelling.* PhD thesis, Vrije Universiteit, Brussel, 2011.

[65] K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944.

[66] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[67] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han. On the Variance of the Adaptive Learning Rate and Beyond. *International Conference on Learning Representations*, 2020.

[68] L. Ljung. Perspectives on System Identification. *Annual Reviews in Control*, 34(1):1–12, 2010.

[69] L. Ljung. *System Identification: Theory for the User.* Prentice Hall information and system sciences series. Prentice Hall PTR, Upper Saddle River, NJ, 2. ed., 14. printing edition, 2012.

[70] L. Ljung, C. Andersson, K. Tiels, and T. B. Schön. Deep Learning and System Identification. In *21st IFAC World Congress*, 2020.

[71] L. Ljung, Q. Zhang, P. Lindskog, and A. Juditski. Estimation of grey box and black box models for non-linear circuit data. *IFAC Proceedings Volumes*, 37(13):399–404, 2004.

[72] J. Lunze. *Regelungstechnik 2: Mehrgrößensysteme, Digitale Regelung.* Springer-Lehrbuch. Springer Berlin Heidelberg, 2016.

[73] A. M. Lyapunov. The general problem of the stability of motion. *International Journal of Control*, 55(3):531–534, 1992.

[74] D. W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[75] D. Masti and A. Bemporad. Learning Nonlinear State-Space Models Using Deep Autoencoders. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 3862–3867, 2018.

[76] M. Matthews and G. Moschytz. The Identification of Nonlinear Discrete-Time Fading-Memory Systems Using Neural Network Models. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 41(11):740–751, 1994.

[77] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[78] T. McKelvey, H. Akçay, and L. Ljung. Subspace-Based Multivariable System Identification from Frequency Response Data. *Automatica*, 32(6):885–902, 1996.

[79] T. McKelvey, A. Helmersson, and T. Ribarits. Data driven local coordinates for multivariable linear systems and their application to system identification. *Automatica*, 40(9):1629–1635, 2004.

[80] A. Miranian and K. Rouzbehi. Nonlinear Power System Load Identification Using Local Model Networks. *IEEE Transactions on Power Systems*, 28(3):2872–2881, 2013.

[81] P. I. Miu and H.-D. Kutzbach. Modeling and simulation of grain threshing and separation in threshing units – Part I. *Computers and Electronics in Agriculture*, 60(1):96–104, 2008.

[82] J. J. Moré. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[83] T. Münker. *Machine Learning with Finite Impulse Response Models.* PhD thesis, University of Siegen, 2021.

[84] T. Münker, G. Kampmann, M. Schüssler, and O. Nelles. System Identification and Control of a Polymer Reactor. *IFAC-PapersOnLine (21st IFAC World Congress)*, 53(2):437–442, 2020.

[85] R. Murray-Smith and T. A. Johansen. *Local Learning in Local Model Networks*, chapter 7, pages 185–210. Taylor and Francis, 1997.

[86] R. Murray-Smith and T. A. Johansen. *Multiple Model Approaches to Modelling and Control.* Taylor and Francis systems and control book series. Taylor and Francis, London, UK, 1997.

[87] K. S. Narendra and K. Parthasarathy. Identification and Control of Dynamical Systems Using Neural Networks. *IEEE Transactions on neural networks*, 1(1):4–27, 1990.

[88] O. Nelles. Axes-Oblique Partitioning Strategies for Local Model Networks. In *2006 IEEE International Symposium on Intelligent Control*, pages 2378–2383, Oct. 2006.

[89] O. Nelles. *Nonlinear System Identification.* Springer International Publishing, 2 edition, 2020.

[90] O. Nelles and R. Isermann. Basis Function Networks for Interpolation of Local Linear Models. In *35th IEEE Conference on Decision and Control*, pages 470–475, 1996.

[91] O. Nerrand, P. Roussel-Ragot, L. Personnaz, G. Dreyfus, and S. Marcos. Neural networks and nonlinear adaptive filtering: Unifying concepts and new algorithms. *Neural computation*, 5(2):165–199, 1993.

[92] O. Nerrand, P. Roussel-Ragot, D. Urbani, L. Personnaz, and G. Dreyfus. Training Recurrent Neural Networks: Why and How? An Illustration in Dynamical Process Modeling. *IEEE Transactions on Neural Networks*, 5(2):178–184, 1994.

[93] J.-P. Noël, A. F. Esfahani, G. Kerschen, and J. Schoukens. A nonlinear state-space approach to hysteresis identification. *Mechanical Systems and Signal Processing*, 84:171–184, 2017.

[94] J.-P. Noël and M. Schoukens. Hysteretic benchmark with a dynamic nonlinearity. In *Workshop on Nonlinear System Identification Benchmark*, Apr. 2016.

[95] O. Ogunmolu, X. Gu, S. Jiang, and N. Gans. Nonlinear Systems Identification Using Deep Dynamic Neural Networks. *arXiv preprint arXiv:1610.01439*, 2016.

[96] J. Paduart. *Identification of Nonlinear Systems using Polynomial Nonlinear State Space Models.* PhD thesis, Vrije Universiteit, Brussel, 2008.

[97] J. Paduart, G. Horvath, and J. Schoukens. Fast identification of systems with nonlinear feedback. *IFAC Proceedings Volumes*, 37(13):381–385, 2004.

[98] J. Paduart, L. Lauwers, J. Swevers, K. Smolders, J. Schoukens, and R. Pintelon. Identification of nonlinear systems using Polynomial Nonlinear State Space models. *Automatica*, 46(4):647–656, 2010.

[99] T. J. Peter, M. Schüssler, D. Rüschen, and O. Nelles. Identification of a potable water system. *at – Automatisierungstechnik*, 69(10):870–879, 2021.

[100] R. Pintelon. Frequency Domain Subspace System Identification Using Non-Parametric Noise Models. *Automatica*, 38(8):1295–1311, 2002.

[101] R. Pintelon and J. Schoukens. *System Identification: A Frequency Domain Approach*. John Wiley & Sons, 2012.

[102] X. Qian, H. Huang, X. Chen, and T. Huang. Generalized Hybrid Constructive Learning Algorithm for Multioutput RBF Networks. *IEEE Transactions on Cybernetics*, 47(11):3634–3648, 2017.

[103] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[104] S. J. Reddi, S. Kale, and S. Kumar. On the Convergence of ADAM and Beyond. In *International Conference on Learning Representations*, 2018.

[105] R. Relan, K. Tiels, A. Marconato, and J. Schoukens. An Unstructured Flexible Nonlinear Model for the Cascaded Water-tanks Benchmark. *IFAC-PapersOnLine*, 50(1):452–457, 2017.

[106] I. Rivals and L. Personnaz. Black-Box Modeling with State-Space Neural Networks. In *Neural Adaptive Control Technology*, pages 237–264. World Scientifix, Apr. 1996.

[107] J. Roll. *Local and Piecewise Affine Approaches to System Identification*. PhD thesis, Linköpings Universitet, Linköping, Sweden, 2003.

[108] F. Rosenblatt. *Principles of neurodynamics: perceptions and the theory of brain mechanisms*. Spartan Books, 1962.

[109] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[110] C. Rudin and J. Radin. Why Are We Using Black Box Models in AI When We Don't Need To? A Lesson From An Explainable AI Competition. *Harvard Data Science Review*, 1(2), Nov. 2019.

[111] F. Sabahi and M. R. Akbarzadeh-T. Extended Fuzzy Logic: Sets and Systems. *IEEE Transactions on Fuzzy Systems*, 24(3):530–543, 2016.

[112] L. E. Scales. *Introduction to Non-linear Optimization*. Macmillan Education UK, 1985.

[113] S. Schaller. Erstellung eines physikalischen Modells für einen Industrie-Dieselmotor zur Nutzung in einer Model-in-the-Loop Simulationsumgebung. Master's thesis, RWTH Aachen University, 2018. Unpublished.

[114] T. B. Schön, A. Wills, and B. Ninness. System identification of nonlinear state-space models. *Automatica*, 47(1):39 – 49, 2011.

[115] J. Schoukens and J. Decuyper. Decoupling in Black Box Nonlinear System Identification: a Game Changer. *19h IFAC Symposium on System Identification SYSID 2021*, 2021.

[116] J. Schoukens, T. Dobrowiecki, and R. Pintelon. Parametric and Nonparametric Identification of Linear Systems in the Presence of Nonlinear Distortions-A Frequency Domain Approach. *IEEE Transactions on Automatic Control*, 43(2):176–190, 1998.

[117] J. Schoukens and L. Ljung. Nonlinear System Identification: A User-Oriented Road Map. *IEEE Control Systems*, 39:28–99, 2019.

[118] J. Schoukens, M. Vaes, and R. Pintelon. Linear system identification in a nonlinear setting: Nonparametric analysis of the nonlinear distortions and their impact on the best linear approximation. *IEEE Control Systems Magazine*, 36(3):38–69, 2016.

[119] M. Schoukens and F. Griesing Scheiwe. Modeling Nonlinear Systems Using a Volterra Feedback Model. In *Workshop on Nonlinear System Identification Benchmark*, Apr. 2016.

[120] M. Schoukens, P. Mattsson, T. Wigren, and J.-P. Noël. Cascaded tanks benchmark combining soft and hard nonlinearities. *Workshop on Nonlinear System Identification Benchmarks, Brussels, Belgium*, pages 20–23, 2016.

[121] M. Schoukens and J.-P. Noël. Three Benchmarks Addressing Open Challenges in Nonlinear System Identification. *IFAC-PapersOnLine (20th IFAC World Congress)*, 50(1):446 – 451, 2017.

[122] M. Schoukens and K. Tiels. Identification of Block-Oriented Nonlinear Systems starting from Linear Approximations: A Survey. *Automatica*, 85:272–292, 2017.

[123] M. Schüssler, T. Münker, and O. Nelles. Deep Recurrent Neural Networks for Nonlinear System Identification. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 448–454, Dec. 2019.

[124] M. Schüssler, T. Münker, and O. Nelles. Local Model Networks for the Identification of Nonlinear State Space Models. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 6437–6442, Dec. 2019.

[125] M. Schüssler and O. Nelles. Extrapolation Behavior Comparison of Nonlinear State Space Models. *IFAC-PapersOnLine (19th IFAC Symposium on System Identification SYSID 2021)*, 54(7):487–492, 2021.

[126] M. Schüssler and O. Nelles. Optimization Approaches for Nonlinear State Space Models. *IEEE Control Systems Letters*, 5(4):1375–1380, 2021.

[127] D. F. Shanno. Conditioning of Quasi-Newton Methods for Function Minimization. *Mathematics of Computation*, 24(111):647–656, 1970.

[128] R. Shorten and R. Murray-Smith. Side-Effects of Normalising Basis Functions in Local Model Networks. In R. Murray-Smith and T. J. (Eds.), editors, *Multiple Model Approaches to Modelling and Control*, chapter 8, pages 211–229. Taylor & Francis, London, 1997.

[129] J. Sjöberg. On estimation of nonlinear black-box models: how to obtain a good initialization. In *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop*, pages 72–81, 1997.

[130] J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.-Y. Glorennec, H. Hjalmarsson, and A. Juditsky. Nonlinear Black-Box Modeling in System Identification: a Unified Overview. *Automatica*, 31(12):1691–1724, 1995.

[131] V. Smits, M. Schüssler, G. Kampmann, C. Illg, T. Decker, and O. Nelles. Modeling Benchmark on NOx Emissions of a Diesel Engine excited by an APRBS inside a Non-Convex Hull. In *6th IEEE Conference on Control Technology and Applications*, 2022. Accepted for publication.

[132] L. Sragner, J. Schoukens, and G. Horváth. Modelling of a slightly nonlinear system: a neural network approach. *IFAC Proceedings Volumes (6th IFAC Symposium on Nonlinear Control Systems)*, 37(13):387–392, 2004.

[133] A. Svensson and T. B. Schön. A flexible state–space model for learning nonlinear dynamical systems. *Automatica*, 80:189–199, 2017.

[134] K. Tanaka and M. Sugeno. Stability analysis and design of fuzzy control systems. *Fuzzy Sets and Systems*, 45(2):135–156, 1992.

[135] K. Tiels. A Polynomial Nonlinear State-Space Matlab Toolbox. In *European Research Network System Identification (ERNSI) Workshop*, 2015.

[136] A. van Mulders, J. Schoukens, and L. Vanbeylen. Identification of systems with localised nonlinearity: From state-space to block-structured models. *Automatica*, 49(5):1392–1396, 2013.

[137] J. A. Vargas, W. Pedrycz, and E. M. Hemerly. Improved Learning Algorithm for Two-Layer Neural Networks for Identification of Nonlinear Systems. *Neurocomputing*, 329:86 – 96, 2019.

[138] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[139] V. Verdult. *Nonlinear System Identification : A State-Space Approach*. PhD thesis, University of Twente, Netherlands, 2002.

[140] V. Verdult. Identification of Local Linear State-Space Models: The Silver-Box Case Study. *IFAC Proceedings Volumes*, 37(13):393–398, 2004.

[141] J. Wang, A. Hertzmann, and D. J. Fleet. Gaussian Process Dynamical Models. In *Advances in Neural Information Pocessing Systems*, pages 1441–1448, 2006.

[142] P. Welch. The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics*, 15(2):70–73, 1967.

[143] D. Westwick, G. Hollander, and J. Schoukens. The Decoupled Polynomial NARX Model: Parameter Reduction and Structural Insights for the Bouc-Wen Benchmark. In *Workshop on Nonlinear System Identification Benchmarks*, 2017.

[144] T. Wigren and J. Schoukens. Three free data sets for development and benchmarking in nonlinear system identification. In *2013 European Control Conference (ECC)*, pages 2933–2938, July 2013.

[145] A. Wills and B. Ninness. On Gradient-Based Search for Multivariable System Estimates. *IEEE Transactions on Automatic Control*, 53(1):298–306, 2008.

[146] K. Worden. Is System Identification Just Machine Learning? *Workshop on Nonlinear System Identification Benchmarks, Brussels, Belgium*, 2016. Unpublished.

[147] L. Zadeh. On the Identification Problem. *IRE Transactions on Circuit Theory*, 3(4):277–281, 1956.

[148] M. Zaheer, S. Reddi, D. Sachan, S. Kale, and S. Kumar. Adaptive Methods for Nonconvex Optimization. In *Advances in Neural Information Processing Systems 31*, pages 9793–9803. Curran Associates, Inc., 2018.

[149] S. Zahn. *Arbeitsspielaufgelöste Modellbildung und Hardware-in-the-Loop-Simulation von PKW-Dieselmotoren mit Abgasturboaufladung.* PhD thesis, TU Darmstadt, Darmstadt, 2012.

[150] J. M. Zamarreño and P. Vega. State space neural network. Properties and application. *Neural Networks*, 11(6):1099–1112, 1998.

[151] M. Zeitz. Canonical Forms for Nonlinear Systems. *IFAC Proceedings Volumes (Nonlinear Control Systems Design 1989)*, 22(3):33–38, 1989.

[152] R. Zimmerschied. *Identifikation nichtlinearer Prozesse mit dynamischen lokal-affinen Modellen: Maßnahmen zur Reduktion von Bias und Varianz.* PhD thesis, Technische Universität Darmstadt, Institut für Automatisierungstechnik, 2008.

[153] R. Zimmerschied and R. Isermann. Regularisierungsverfahren für die Identifikation mittels lokal-affiner Modelle (Regularization Techniques for Identification Using Local-Affine Models). *at – Automatisierungstechnik*, 56(7):339–349, 2008.

[154] R. Zimmerschied and R. Isermann. Nonlinear System Identification of Block-oriented Systems using Local Affine Models. *IFAC Proceedings Volumes (15th IFAC Symposium on System Identification)*, 42(10):658–663, 2009.