Scientific machine learning for hybrid modeling in real world engineering: Validated on examples from automotive and medical industry

Tobias Thummerer

Angaben zur Veröffentlichung / Publication details:

Thummerer, Tobias. 2026. “Scientific machine learning for hybrid modeling in real world engineering: Validated on examples from automotive and medical industry.” Augsburg: Universität Augsburg.

Scientific Machine Learning for Hybrid Modeling in Real World Engineering

Validated on Examples from Automotive and Medical Industry

Tobias Thummerer

Dissertation
zur Erlangung des Doktorgrades
Dr. rer. nat.

Fakultät für Angewandte Informatik
Institut für Informatik
Universität Augsburg

10.11.2025

Erstgutachter*in:	Prof. Dr.-Ing. Lars Mikelsons
Zweitgutachter*in:	Prof. Dr. Jörg Hähner
Drittgutachter*in:	Assoc. Prof. Mohamed Tahar Mabrouk, Ph.D.
Tag der mündlichen Prüfung:	16. März 2026

Abstract

Scientific machine learning (SciML), the combination of scientific computing methods and machine learning, is a rapidly growing field of research with a huge potential. An almost daily growing amount of scientific publications shows, that methods from this field are able to cope challenges, that were not — or at least hardly — solvable with methods from the repertoire of classical engineering under similar conditions. However, it is further a *very young* field of research and at the time of writing, it is missing *industrial robustness* with regard to various aspects. One of the most relevant — and one of the core topics in this thesis — is the seamless integration of models of industrial scale as part of *hybrid modeling*, the fusion of models from classical engineering and models from machine learning. Within this work, models of *industrial scale* are characterized by systems of equations consisting of hundreds to thousands of (different) equations, that may further be numerically challenging to solve and include complexities like discontinuities, algebraic loops and discrete subsystems — topics, that are often neglected in state-of-the-art literature. In this work, we examine these topics in detail, derive requirements, and address the underlying challenges with new methods. Mathematically, we extend state-of-the-art hybrid modeling so that it is capable of describing common physics-based simulation models, machine learning models, as well as combinations of both. We derive rules of combination for this hybrid modeling class and methods for initialization, simulation, training and regularization. These new methods are illustrated at seven experiments, using a recurring academic example model.
After these theoretical considerations and derivations, we finally construct the *neural Functional Mock-up Unit*, a technical implementation of the developed hybrid modeling class, and conclude this thesis with the detailed examination of three successfully processed real-world use cases, two of them from industry, all involving challenging simulation models. We show in a critical validation, that the proposed methods and the resulting hybrid models are able to outperform pure physics-based state-of-the-art simulation models and are even able to solve problems, that are hardly — or not at all — solvable by pure machine learning models.

Dedication

This work is dedicated to my beloved wife *Patrizia*, who always summarizes my work for others very efficiently with the words:

"He's doing things with neural networks."

And this is scientifically — as you will see — correct. You are my activation function and even if you run me into saturation sometimes, I know that life — just like neural networks — becomes quite boring without adding a little nonlinearity.

Acknowledgment

I want to thank my parents, my mother *Veronika*, who was and still is always supporting my ideas and decisions in life. I want to thank my father *Konrad*, who passed on my first computer to me at young age together with a C/C++ IDE, and who has always encouraged my interest in natural sciences in a variety of ways. And my sister *Anja* and my brother *Dominik*, that prevent me from diving to deep into my work and get me back to the other (probably more) important things in life.

I want to thank my boss and academic supervisor *Lars* who has never restricted me in my ideas. This is anything but a matter of course. Let's keep going like this! I would also like to thank the hard-working chair secretary *Heike*, who guides us safely through the bureaucratic jungle and "keeps the machine running". Without you two — our *chair parents* — the wonderful time as a doctoral student including many impressive research stays abroad would not have been possible.

Further, I want to thank all my colleagues from the *Chair of Mechatronics* for creating such a friendly working environment and for the productive verbal exchanges. My special thanks go to the brave ones who shared (some time) the office with me: *Lennart*, *Peter*, *Frederic*, and last but not least, *Josef*, who — together with *Julian* — reminded me exactly 45 times that I had exceeded my planned deadline by another week. I think it made a difference. Maybe. In any case, I will repay this support in kind.

Last but not least I want to thank all partners on the industrial side, especially *Dr. Fabian Jarmolowitz* and *Daniel Sommer* (Robert Bosch GmbH), *Dr. Julia Gundermann* and *Dr. Artem Kolesnikov* (ESI Group), *Prof. Dr. Olaf van der Sluis*, *Dr. Sergei Shulepov* and *Dr. Marco Baragona* (Philips). You challenged the presented methods with your demanding applications and therefore helped them to grow and made them robust enough to hopefully survive in the hard, industrial application world.

Finally, I want to thank all colleagues in, and financiers of the research projects I was or am still working in: UPSIM[1], PHyMoS[2], and OpenSCALING[3].

Thank you all!

[1]ITEA3 #19006 (`https://upsim-project.eu/`)
[2]Bundes Ministerium für Bildung und Forschung (BMBF) (`https://phymos.de/`)
[3]ITEA4 #22013 (`https://openscaling.org/`)

List of Figures

List of Tables

List of Abbreviations

0D zero-dimensional

1D one-dimensional

2D two-dimensional

3D three-dimensional

ABS anti-lock braking system

AD automatic differentiation

AE absolute error

AI artificial intelligence

AE algebraic equation

AML acute myeloid leukemia

ANN artificial neural network

BB bouncing ball

BDF backward differentiation formula

BLT block lower triangular

BMBF Bundes Ministerium für Bildung und Forschung

CAD computer aided design

CB collision border

CFD computational fluid dynamics

CNN convolutional neural network

CPU central processing unit

CS co-simulation

DAE differential algebraic equation

DB derivative-based

DE differential equation

DFT direct feed through

ECM electronically commutated motor

FFNN feed-forward neural network

FMI functional mock-up interface

FMU functional mock-up unit

FNO Fourier neural operator

GUI graphical user interface

GPU graphics processing unit

GRU gated recurrent unit

GT ground truth

HM hybrid model

HNN Hamiltonian neural networks

HPC high performance computing

HUDA mixed hybrid universal discrete algebraic

HVAC heating, ventilation, and air conditioning

IP intellectual property

IPB integrated power brake

IVP initial value problem

JIT just-in-time

JVP Jacobian-vector-product

LSTM long short-term memory

LT lower triangular

MAE mean absolute error

ME model exchange

ML machine learning

MSE mean squared error

NODE neural ordinary differential equation

ODE ordinary differential equation

OEM original equipment manufacturer

ONNX open neural network exchange

OpenSCALING Open standards for SCALable virtual engineerING and operation

PCS pig cardiovascular system

PeN physics-enhanced neural

PDE partial differential equation

PHyMoS Proper Hybrid Models for Smarter Vehicles

PINN physics-informed neural network

PM physical model

PWM pulse-width modulation

QoI quantity of interest

RBFN radial basis function network

RNN recurrent neural network

SA sensitivity analysis

SB solution-based

SCARA selective compliance assembly robot arm

SciCo scientific computing

SciML scientific machine learning

SD standard deviation

SE scheduled execution

SiL software in the loop

SODEF stable neural ODE for defending against adversarial attack

SOP second order preservation

SSDB solution-synchronized derivative-based

SSP system structure and parameterization

STEER simple temporal regularization

TCP tool center point

UA universal approximator

UDE universal differential equation

UODE universal ordinary differential equation

UPSIM Unleash Potentials in SIMulation

VAE variational auto-encoders

VJP vector-Jacobian-product

XML extensible markup language

Contents

Preface

In 2019, only a few months before I started working as a doctoral student, the famous computer scientist *Rich S. Sutton* published the following on his homepage, critically concerning building in *how we think we think* (or *human knowledge*) into machine learning methods:

> *"This is a big lesson. As a field, we still have not thoroughly learned it, as we are continuing to make the same kind of mistakes. To see this, and to effectively resist it, we have to understand the appeal of these mistakes. We have to learn the bitter lesson that building in how we think we think does not work in the long run. The bitter lesson is based on the historical observations that (1) AI researchers have often tried to build knowledge into their agents, (2) this always helps in the short term, and is personally satisfying to the researcher, but (3) in the long run it plateaus and even inhibits further progress, and (4) breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning. The eventual success is tinged with bitterness, and often incompletely digested, because it is success over a favored, human-centric approach."*
>
> - *Rich S. Sutton* [1]

Nevertheless, in this doctoral thesis, we still have not learned the bitter lesson and will continue to make the same kind of mistakes: We will build in human knowledge into machine learning methods and show that this is *not* only satisfying for the researcher, but also helps in the short *and* the long term.

The key point here is that building in human knowledge in machine learning agents (or methods in general) limits the expressiveness of the resulting model. In the *short term*, this allows for fitting the model faster and with less data — which is indeed satisfying to the researcher. In the *long run* an unrestricted model could have learned for the same (and perhaps even better) results, but would also require more data for this task.

Even if we investigate especially the incorporation of physical laws in this thesis, which is perhaps the most critically validated human knowledge we have, I personally agree with this long-term statement for *some* applications in machine learning. Nevertheless, I want to raise one question regarding a (not that small) subset of other real-world applications:

What if there is just not enough data[4] available?

[4]Or from the point of view of capitalism: *Not enough data with reasonable financial effort.*

Chapter 1

Introduction

The scientific discipline of *forecasting* or *time series prediction* relies on data from the past to predict the (near) future. The big problem with data from the past is that it is always limited by two points in time: The time someone decided to start recording data (and hopefully later share it), and *now*. The amount of available data is therefore finite — and sometimes this finite set might be *too small* w.r.t. a specific application.

1.1 Motivation

Machine learning (ML) models in general require large amounts of data. This *large amount* basically depends on the complexity and size of the model, which roughly translates into the number of parameters. In addition to the number of parameters, the general question of how much data is needed *exactly* depends on further factors such as the complexity of the learned function and the quality of the available data.

If the amount of available data is not enough, one can do additional experiments and wait for the results. Doing more experiments and/or waiting is not always an option for different reasons. There are applications that have — and still will have in the future — only access to a very limited amount of data. Two examples from medical and automotive research are:

Patient-specific (tumor) modeling Although we are gaining new insights and getting better every day, cancer is still an illness that is far from being fully understood. For example in case of acute myeloid leukemia (AML)[1], recovery progression is highly variable and therefore difficult to predict under treatment, see Sauerer *et al.* [2]. ML models have the potential to interpret undiscovered correlations and make better predictions about the course of treatment, however, require data for this task. Unfortunately, the most informative data is gathered by bone marrow examinations, which are extremely stressful for the patient and must therefore be kept as low as possible in number.

Prototyping in (automotive) engineering Physical prototypes are still very costly to produce, maintain and operate. As a consequence, the trend of the last years is to substitute prototypes by more economically computer simulations where possible [3]. The creation and validation of such simulation

[1] We are looking forward to starting a research project in this direction together with the *Universitätsklinkium Augsburg* in 2025.

models, so-called *virtual prototypes*, requires data on the other hand. Therefore, it is the economic goal to create valuable simulation models with as little data as possible.

In addition to these two examples, many other applications drastically suffer from limited and too small data available. Luckily, there are four well-discussed strategies to work with models, that are *too large* for the available amount of data:

Synthetic data In case the amount of *real* data — so measurements from the real world system to be modeled — is not sufficient, *synthetic* data can be generated, for example based on computer simulations[2]. This requires, of course, that the process to be learned is understood — at least in significant parts — in order to build a virtual model for data generation. For the sake of completeness, also ML [4] can be used to generate synthetic data, but again requires data for this task in the first place — so a chicken-and-egg problem.

Data augmentation Similarly to synthetic data, existing data can be *augmented*, meaning it is transformed to generate new data samples, instead of generating new data from the scratch. The type of transformation depends on the application domain, in image processing for example (where data augmentation is very common) typical transformations are the rotation, scaling or cropping of existing images to generate new data samples. However, in general, augmented data is less valuable and can be used e.g. to reduce over-fitting, however, does not contain any genuinely new information.

Transfer learning For some applications, models can be *pre-trained* on similar data, that is more easily available and later be *fine-tuned* on the actual limited dataset. Of course, it is crucial that the pre-training and fine-tuning applications are very similar to apply this strategy. One main area of application are language or visual applications, in which so-called *foundation models* that where pre-trained on large datasets, are fine-tuned afterward for a specific task [5].

Induce knowledge If the model is too large regarding the available data, the model can be downsized by reducing the number of parameters. Of course, this also reduces the expressiveness of the model, which is undesirable. To maintain the capability of the model to express the mathematical function we want it to approximate, we can incorporate parts of this function that are already known into our model. Or from a more abstract point of view: Because the underlying goal of machine learning is to extract knowledge from data, we can reduce the amount of needed data by deploying *prior knowledge* to the task of machine learning, that does not need to be extracted from data anymore.

Unfortunately, the first three options are not applicable if the process being learned is not well understood, as can be exemplified at the AML use case. Here, the generation of synthetic data is not possible, because this requires an excellent understanding of the physical — so biological and chemical — processes behind the

[2]One could definitely discuss if this also falls under the category of *built-in human knowledge*, because these simulations are in general based on the experience of a human modeling engineer.

data (so the progression of the disease), which is a topic still to be explored in this case. Data can not be augmented easily for the same reason, for example it is not obvious how genome sequences can be transformed while still reflecting the details of the disease we want to learn for. However, data augmentation is possible and well-discussed for other medical application domains like image processing [6]. And also transfer learning with similar data is not (or at least very limited) applicable here, because there is no (known) disease similar enough with significantly better data access. Furthermore, medical data is in general very sensitive regarding privacy and therefore there is no open cross-patient database for AML, that would enable transfer learning.

For many real world applications, only one option remains: The incorporation of knowledge into the model, which is the core discipline of *scientific machine learning*. If this knowledge consists of physical relationships and is structurally incorporated in the form of mathematical equations, this is referred to as *hybrid modeling*. Both research fields are introduced in the next sections, and an abstract comparison is given in Fig. 1.1.

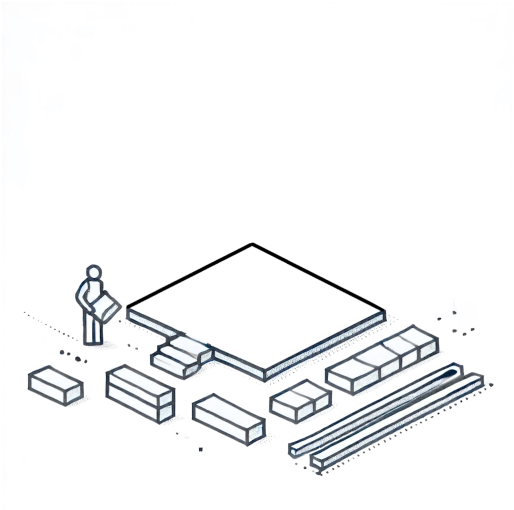
1.2 Hybrid modeling

Having started over 30 years ago [7], *hybrid modeling* refers to the structural combination of physical (or mechanistic) models and machine learning models. Accordingly, the two fields of research to introduce hybrid modeling are *physical modeling* and *machine learning* (ML), compare Fig. 1.2. These two research areas are introduced in more detail in Sec. 2.2 and Cha. 3, and briefly below:

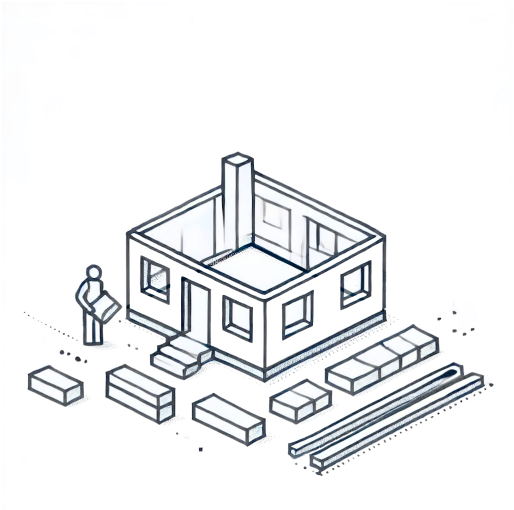
Physical modeling is an important sub-area of scientific computing (SciCo) and deals with the process of finding mathematical equations that describe the physical process to be modeled, concerning given requirements, for example, a predefined limit for the model deviation. An important insight is that physical modeling is always performed based on assumptions that lead to model simplifications and therefore induce a *modeling error* to every physical simulation model. This modeling error is sometimes referred to as *sim-to-real-gap*.

Machine learning (ML) focuses on algorithms which *learn* from data and are therefore capable of extracting knowledge from data. ML is a major part of AI. Training methods can be classified into supervised, unsupervised, and reinforcement learning. Supervised training can be divided into two core problems: Regression (or prediction) and classification. In this thesis, mainly supervised regression problems are investigated. Basic principles of ML are assumed to be prior knowledge of the reader, and a detailed introduction to ML is omitted here.

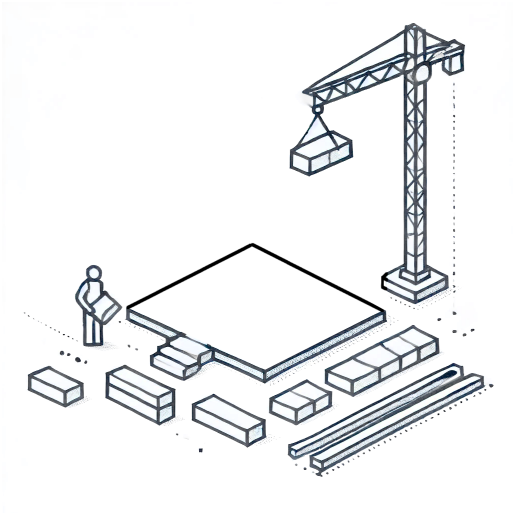
The goal behind hybrid modeling, the symbiosis of these two fields, is to obtain a *better* model, usually in terms of prediction accuracy or simulation performance, e.g. shorter execution time or less hardware requirements. Hybrid modeling is a real subset of SciML and can, in a historical context, even be interpreted as its early foundation. Hybrid modeling is introduced, discussed, and the state of the art is extended in Cha. 3.



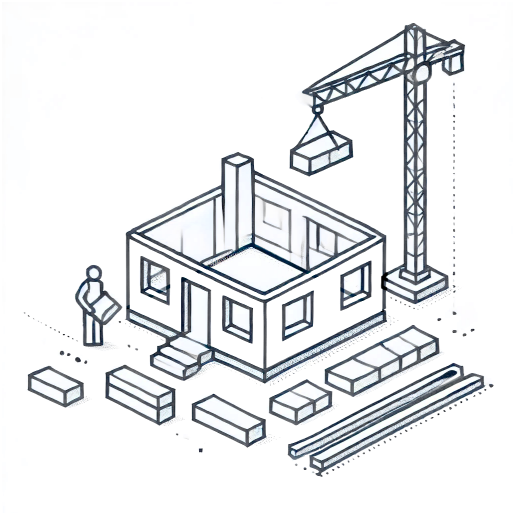
(a) The little person (*machine learning optimizer*) is entrusted with the overwhelming task of building a house (*training a simulation model*). Fortunately, he receives support in various ways ...



(b) Doing **hybrid modeling**, he can start with a solid *model foundation*, which has the direct consequence that less has to be built and less can be done wrong.



(c) With methods from **SciML**, it does not necessarily make the work any less, but it is much easier to build a house with the help of a construction crane (e.g. numerical solvers from SciCo).



(d) Finally, in addition to doing only **hybrid modeling**, multiple methods of **SciML** can be combined to optimize the construction project (in terms of speed as well as quality).

Figure 1.1: Building a simulation model with SciML is (almost) like building a house. These images were generated with the assistance of artificial intelligence (AI). As can be seen from the non-perspective lines, unfortunately no SciML was applied for image generation.

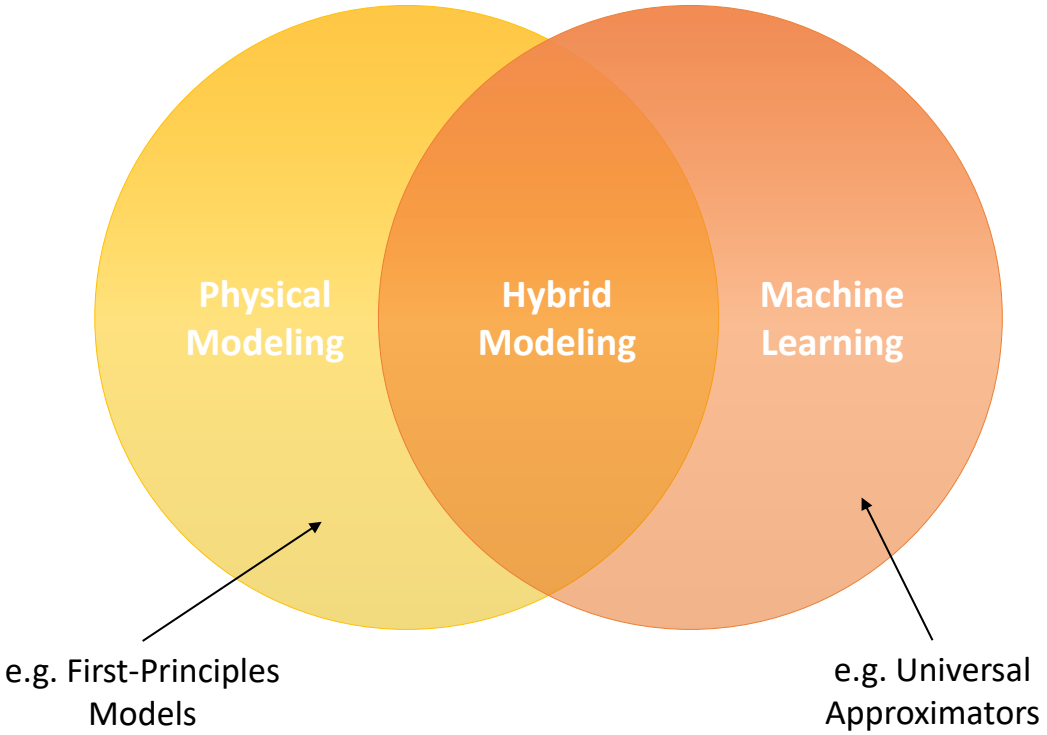


Figure 1.2: Hybrid modeling: The *model-driven* point of view, investigating how models from the physical modeling and machine learning world can be combined.

As introduced, hybrid modeling focuses especially on the *modeling* perspective of this symbiosis. However, around 2019 [8], one took the next step by doing one step back and investigated not only *physical models* as partner for ML, but the entire field of *methods* from SciCo (of which *physical modeling* is a part) — which can be seen as the birth of SciML. Investigating methods instead of only models is a striking difference that can be explained in the historical context: At the beginning of hybrid modeling research, the machine learning part, at this time often a feed-forward neural network (FFNN), was trained in isolation [9]. In SciML on the other hand, advanced methods for sensitivity analysis (e.g. *adjoint methods*, s. Sec. 2.1.3.4) can be applied that allow for optimization of the fully integrated model including the numerical solver and other algorithms, for example. SciML is introduced in the following section.

1.3 Scientific machine learning (SciML)

As already noted, the terms *hybrid modeling* and *scientific machine learning* are not separable easily, and they are indeed often used synonymously. Even if this is not correct, at least it states that *hybrid modeling* can be interpreted as real subpart of SciML and the statement *"I am researching in SciML"* is not wrong but only less precise, if *hybrid modeling* is actually meant.

To introduce SciML, the union of scientific computing (SciCo) and ML, we first need to introduce SciCo itself: The domain of SciCo, often also referred to as *compu-*

tational science, deals with the solution of physical problems (often based on physical models) using mathematical (often numerical) algorithms. The efficient implementation and execution of these algorithms on a computer system plays a key role in SciCo. This especially captures the solving (referred to as *simulation*) of the already mentioned physical system models. If we compare this with the definition of *hybrid modeling*, this results in an extended field of research, shown in Fig. 1.3.

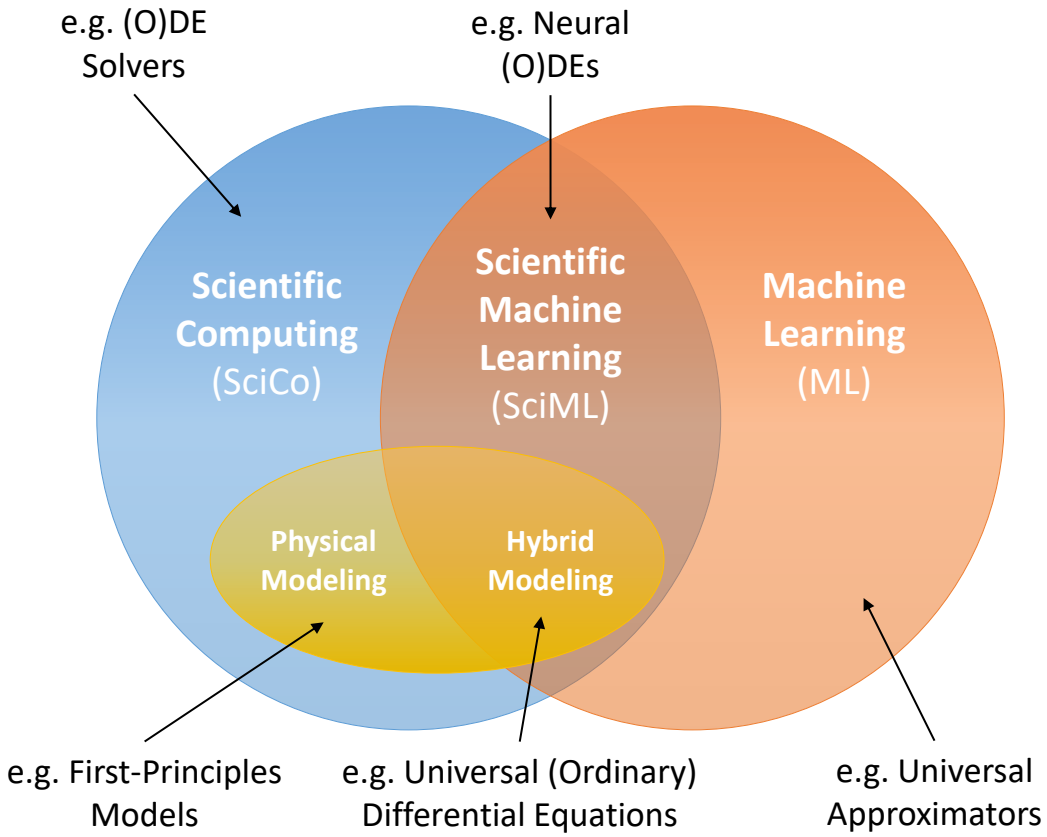


Figure 1.3: Scientific machine learning (SciML): The *method-driven* point of view. An overview of the scientific disciplines involved in this thesis: *Scientific computing* (including *physical modeling*) and *machine learning*. The intersections and main fields of contribution are *hybrid modeling* and *scientific machine learning*.

The term scientific machine learning was established within the technical report *Basic Research Needs Workshop for Scientific Machine Learning* [8] and further summarized by the *SciML Research Group* at *Brown University* with the following lines:

"Scientific machine learning (SciML) is an emerging discipline within the data science community. SciML seeks to address domain-specific data challenges and extract insights from scientific data sets through innovative methodological solutions. SciML draws on tools from both machine learning and scientific computing to develop new methods for scalable, domain-aware, robust, reliable, and interpretable learning and data analysis, and will be critical in driving the next wave of data-driven scientific discovery in the physical and engineering sciences.

Like scientific computing, SciML is multidisciplinary and leverages expertise from applied and computational mathematics, computer science, and the physical sciences."

- SciML Research Group at Brown University [10]

The four major key messages from this to take away for reading this thesis are:

Interdisciplinarity SciML is very interdisciplinary and aims at creating new methods based on machine learning and scientific computing (e.g. applied mathematics, computer and physical sciences).

Explainability The goal of SciML is not only to learn behavior from scientific data, but also to *extract insights*. This also aims at explainability of scientific phenomena, so SciML is intended to be a tool for *scientific discovery*.

Domain-aware SciML (often) aims at *domain-specific* solutions, meaning it is acceptable to propose a specialized solution for an application domain like e.g. multi-body dynamics. From the point of view of a mathematician this might seem unusual, because in applied mathematics often a class of mathematical problems (like e.g. ordinary differential equations (ODEs)) beyond a specific application domain is investigated. In engineering, on the other hand, it is common to focus on a specific application domain.

Relevance SciML is an emerging field of research and assumed to enable the next generation of scientific discoveries in physics and engineering sciences.

Further, the underlying technical report from Baker *et al.* [8] nicely concludes open research questions, which have still not been definitively answered six years later. We collect the subset of open challenges for SciML, that are faced within this thesis:

Incorporation of domain knowledge into supervised ML [8]
Mathematically, the training of a SciML model is nothing but solving an optimization problem: Finding model parameters, so that the optimization objective is met — or met as good as possible. From the point of view of *optimization*, domain knowledge boils down to an *optimization constraint*. During optimization, we can distinguish between two modes of applying constraints: First, constraints can be guaranteed *hard* in form of either applying a constrained optimization or by choosing a model structure, that satisfies these constraints by design, meaning it is not able to violate the constraints for any given data (within the considered scope). This is for example the case for neural ODEs [11], that incorporate numerical ODE

solvers. Second, constraints can be tried to satisfy (*soft*) by multi-objective optimization, with main objective and the constraints as secondary objectives, or explicit regularization, by extending the main objective function by additional terms. This is for example the case for physics-informed neural networks (PINNs) [12], where a physical model can act as regularization term in the cost function. In this work, we discuss *hard* incorporation of domain knowledge in form of hybrid modeling in Cha. 3, as well as multiple strategies for *soft* regularization in Cha. 6.

Modeling and representation of domain knowledge in SciML [8]
Domain knowledge can intuitively be represented as part of simulation models. However, the integration of simulation models of industrial scale is not trivial, and to this day there is no extensive tool and modeling language for this task available. Probably the most advanced tool for this purpose is the Julia programming language, paired with the library *ModelingToolkit.jl* for acausal modeling [13], [14], and further libraries of the Julia SciML ecosystem (s. Sec. 2.3.2). In this work, we deliberately take a different approach: The general integration of simulation models of industrial scale is discussed in Cha. 3 and a solution via FMI is deduced in Cha. 7.

Keeping the presented open challenges in mind, the overall goal of this thesis can be summarized in the following.

1.4 Goal of this thesis

Summarized in one sentence, this work deals with hybrid model development in the context of SciML, concerning challenges as they appear in real-world modeling. This involves the creation of hybrid simulation models, studied with a strong focus on industrial applicability. We accomplish this by breaking down industrial requirements to a mathematical level as well as by concerning further practical requirements that go beyond math, such as the availability of software functionality. The process of *creation* can be divided into two sub-steps, that mirror the open challenges from Baker *et al.* [8]: First, the architectural construction of the parameterizable hybrid model (*modeling and representation of domain knowledge*), and second, the efficient determination of the implied parameters (*incorporation of domain knowledge*), which explicitly exceeds the boundaries of hybrid modeling and investigates methods from SciML. The overall goal of this thesis is therefore:

To advance the applicability of hybrid simulation models based on SciML in the industrial field, by addressing and solving the challenges that typically emerge during working on different engineering use cases.

Of course, this work deals not with *every* hybrid modeling challenge in the field, but is intended to provide guidance to understand and avoid the usual stumbling blocks as they appear in engineering practice.

1.5 Structure of this thesis

Besides common opening (introduction, basics) and closing chapters (validation, conclusion), the methodological contributions of this thesis are structured in anal-

ogy to the intuitive development process of a hybrid model: The model is *created* and then *initialized*, the model is *trained* and methods of *regularization* are applied to further enhance results. Some *practical* adaptions are made to move from mathematics to the real application. Altogether, this results in a structure, that is briefly introduced in the following:

Introduction The motivation and basic introduction to the topic, including the definition of the terms *hybrid modeling* and *scientific machine learning* is handled within Cha. 1 (this chapter).

Basics After finishing the motivation and introduction to the core topic of this thesis, we investigate mathematical basics, including methods from SciCo, in Cha. 2. Further, because in hybrid modeling (and SciML as well) two different application domains with different domain experts collide, we define some common terminology between both fields.

Creation As introduced, we investigate and extend the state of the art of hybrid modeling in Cha. 3. Very simple, as well as up-to-date architectures for hybrid modeling are investigated and requirements are collected. Based on these, we derive a class of hybrid models for investigation and propose a new method for hybrid model creation and interpretation. We summarize the chapter with the discussion of a related experiment.

Initialization After the general definition of a hybrid model class, we discuss the proper initialization in Cha. 4. This covers initialization of the physical and ML model, as well as the connections in between.

Training After creation and initialization of a hybrid model, we can perform the actual training. In Cha. 5, we adapt common concepts from related work to fit the introduced hybrid modeling class and further adapt and compare methods for training.

Regularization Training of challenging hybrid models is prone to unintended termination or local minima, that can be tackled by advanced *regularization* techniques that we introduce in Cha. 6.

Practice Even if the methods that have been introduced up to this point are derived from the requirements of industrial model, there exist further open challenges regarding technical aspects. In Cha. 7, we deal with the technical applicability of the proposed methods to models that exist in an industry-typical format. This is accomplished by discussing a seamless integration of FMI into the development process.

Validation Equipped with methods and strategies to deal with simulation models of industrial scale, we are able to apply the proposed methodology to three very different and challenging applications from the automotive, medical and robotic industry in Cha. 8: The simulation of an electric motor within an electro-hydraulic power brake, the simulation of the cardiovascular system of a pig (as a placeholder for humans), and the simulation of a robot arm, writing with a pen on a piece of paper.

Conclusion This dissertation ends with the final chapter 9, in which we summarize the contributions and limitations, and complement some directions of current and future research related to the topic.

As announced, we proceed by introducing some basics that are relevant across multiple chapters.

Chapter 2

Basics

Within this chapter, we want to introduce general mathematical concepts that are relevant for the entire thesis, as well as some common terminology. Further, the used programming language and software are introduced, together with a recurring example system that supports understanding of the introduced methodology at various points within this work.

2.1 Mathematical foundation

In this section, we introduce the notation as used within this thesis, basic types of equations, and the topic of local sensitivity analysis (SA) for ODEs.

2.1.1 Notations and conventions

We start by introducing the notation for mathematical structures as is used in this thesis.

Scalars, vectors and matrices
If not stated differently, scalars are always denoted with lowercase Arabic or Greek letters. Vectors are always column vectors, and row vectors are denoted as transposed column vectors, see Tab. 2.1.

Table 2.1: Notation of scalars, vectors and matrices.

Identifier	Dim.	Notation	Example
Scalar	1	lower-case	$a, \alpha \in \mathbb{R}$
Vector (column)	$n \in \mathbb{N}$	lower-case, bold	$\boldsymbol{a}, \boldsymbol{\alpha} \in \mathbb{R}^n$
Vector (row)	$m \in \mathbb{N}$	lower-case, bold	$\boldsymbol{a}^T, \boldsymbol{\alpha}^T \in \mathbb{R}^m$
Super-vector	$n \cdot m \in \mathbb{N}$	upper-case, bold	$\boldsymbol{A} \in \mathbb{R}^{n \cdot m}$
Matrix	$n \times m \in \mathbb{N}^2$	upper-case, bold	$\boldsymbol{A} \in \mathbb{R}^{n \times m}$

Sets and ranges
Further, we can introduce sets and ranges as in Tab. 2.2.

Table 2.2: Notation of sets and ranges.

Identifier	Notation	Example
Sets	curved brackets, comma separated	$\{1, 2, ..., n\}$
(Inclusive) range	square bracket pair, comma separated	$[0, 1]$

Products
Different products are used in this thesis and are listed in Tab. 2.3.

Table 2.3: Different product notations.

Symbol	Description	Example
$\cdot$ (or nothing)	scalar, matrix-vector and matrix product	$\boldsymbol{A} \cdot \boldsymbol{b}$ or $\boldsymbol{Ab}$
$\odot$	Hadamard (element-wise) product	$\boldsymbol{a} \odot \boldsymbol{b}$
$\oslash$	Hadamard (element-wise) division	$\boldsymbol{a} \oslash \boldsymbol{b}$
$\times$	cross product	$\boldsymbol{a} \times \boldsymbol{b}$

Lengths and norms
Length and norms are defined for vectors as shown in Tab. 2.4.

Table 2.4: Lengths and norms.

Symbol	Description	Example
$\lvert$	number of elements (element length)	$\lvert\boldsymbol{a}\rvert$
$\lVert\rVert_1$	one norm of vector (taxicab distance)	$\lVert\boldsymbol{a}\rVert_1$
$\lVert\rVert_2$ or $\lVert\rVert$	two norm of vector (Euclidean distance)	$\lVert\boldsymbol{a}\rVert_2$ or $\lVert\boldsymbol{a}\rVert$

Special structures
Different special vectors and matrices are used, see Tab. 2.5.

Table 2.5: Different special structures.

Symbol	Description
$\mathbf{0}$	vector consisting of zeros
$\mathbf{1}$	vector consisting of ones
$\boldsymbol{I}$	identity matrix
$diag(\boldsymbol{a})$	diagonal matrix with elements of vector $\boldsymbol{a}$ on diagonal

2.1.2 Types of equations

In this thesis, we examine different types of equations that need to be dealt with differently and are therefore introduced very briefly in the following. A detailed introduction can be found within numerous text books concerning numerical solutions of differential equations (DEs) such as Cellier and Kofman [15].

Algebraic equations (AEs)
Within this work, a function $\boldsymbol{g}$ maps from an *input* space $\mathbb{U}$ to an *output* space $\mathbb{Y}$, as in

$$\boldsymbol{g} : \mathbb{U} \rightarrow \mathbb{Y}, \tag{2.1}$$

where $\mathbb{U}$ and $\mathbb{Y}$ are in general the $\mathbb{R}^n$ in this work, but not necessarily sharing the same dimension for n. If we pick an input $\boldsymbol{u} \in \mathbb{U}$ and an output variable $\boldsymbol{y} \in \mathbb{Y}$, we can evaluate the function and get an *algebraic equation*. An algebraic equation expresses the equality of two mathematical statements, like in

$$\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{u}). \tag{2.2}$$

Further, simulation requires the evaluation of a set of equations for different points in time, therefore variables are not time-constant values like in the previous equation, but may change over time — indeed they are time-dependent functions themselves:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{u}(t)). \tag{2.3}$$

For now, there is no computational order assumed. We can solve for $\boldsymbol{y}(t)$ by inserting $\boldsymbol{u}(t)$, but could also invert $\boldsymbol{g}$ (if invertible) and solve for $\boldsymbol{u}(t)$ based on $\boldsymbol{y}(t)$. If an equation is *causalized*, meaning a computational order for the variables is assigned, variables can be referred to as *knowns* (or *inputs*) and *unknowns* (or *outputs*). The causalized equation with known/input $\boldsymbol{u}(t)$ and unknown/output $\boldsymbol{y}(t)$, can be explicitly denoted by

$$\boldsymbol{y}(t) := \boldsymbol{g}(\boldsymbol{u}(t)). \tag{2.4}$$

However, in the following, we proceed using "=" instead of ":=" and distinguish between causalized and non-causalized equations depending on the context.

An algebraic equation might also be *parameterizable*, meaning that there is a set of constant parameters that can be adapted to influence the relation between variables $\boldsymbol{u}$ and $\boldsymbol{y}$, so

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{u}(t), \boldsymbol{p}). \tag{2.5}$$

This equation *implicitly* depends on time, because $\boldsymbol{u}(t)$ changes over time. Algebraic equations can further *explicitly* dependent on time t, as in

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{u}(t), \boldsymbol{p}, t). \tag{2.6}$$

Besides AEs, also differential equations (DEs) are relevant in engineering practice, and are introduced in the following.

Ordinary differential equation (ODE)

An ODE describes the change of one or more quantities in time by defining the *derivative* of these quantities. Therefore, it is an ideal candidate to express dynamic systems in physics. A generic ODE in *state-space* form can be denoted as

$$\dot{\boldsymbol{x}}(t) = \frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t). \tag{2.7}$$

Here, $\boldsymbol{x}$ is the system state vector, $\dot{\boldsymbol{x}}$ is the system state derivative vector, $\boldsymbol{u}$ a vector of inputs (optional), $\boldsymbol{p}$ the parameter vector and t the time. The vector function $\boldsymbol{f}$ is often referred to as the *right-hand side* of the ODE.

Whereas some simple ODEs can be solved analytically for $\boldsymbol{x}(t)$, most systems of ODEs relevant to practice can only be solved numerically by an ODE solver. For a given time span $[t_0, t_f]$ and an initial condition $\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$, the ODE system can be solved numerically. By numerically solving, the numerical ODE solution $\boldsymbol{X} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_n\}$ is only computed for a finite number n of discrete time points t_i,

with $i \in \{0, ..., n\}$. This results in $t_f = t_n$, so the final time equals the last discretized time point. In addition to the discretized solution, many numerical solvers offer an interpolation polynomial that approximates the exact ODE solution for arbitrary points in time within the given solution time span. Even if the solution polynomial is in general not the exact solution of the ODE, we continue to denote it as $\boldsymbol{x}(t)$ and make a distinction between exact and approximated solution clear if necessary.

Probably the most famous (and easiest) numerical solver is the *explicit Euler integration method*, often referred to as *forward Euler*, and is defined as

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h \cdot \boldsymbol{f}(\boldsymbol{x}_i, \boldsymbol{u}_i, \boldsymbol{p}, t_i) \quad \text{for} \quad i \in \{0, ..., n-1\}. \tag{2.8}$$

It can easily be seen that for a given $\boldsymbol{x}_i$ (and given inputs, parameters, and time) the ODE system can be solved for the next step $\boldsymbol{x}_{i+1}$ with solver step size h. Because all values on the right side depend on the point in time t_i and are therefore known, the method is considered *explicit*. To obtain a solution over a given time span, multiple steps can be performed one after another, based on the initial state $\boldsymbol{x}_0$.

Besides the forward Euler method, also an *implicit* version of the algorithm is available, that is defined as

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + h \cdot \boldsymbol{f}(\boldsymbol{x}_{i+1}, \boldsymbol{u}_{i+1}, \boldsymbol{p}, t_{i+1}) \quad \text{for} \quad i \in \{0, ..., n-1\}. \tag{2.9}$$

Obviously, the only difference is the evaluation of $\boldsymbol{f}$ at the *next* point of the state trajectory $\boldsymbol{x}_{i+1}$ (as well as input and time) instead of the current point. Because $\boldsymbol{x}_{i+1}$ appears on both sides of the equation and within the (in general) nonlinear function $\boldsymbol{f}$, it requires solving a non-linear system of equations to obtain $\boldsymbol{x}_{i+1}$. A common choice for this is Newton's method (or optimized versions of it), which features quadratic order of convergence, however, at a high computational cost of $\mathcal{O}(n^3)$, caused by inverting the Jacobian used for successive determination of the solution (zero crossing). Optimized versions exist, however, still require $\mathcal{O}(n^{>2})$. Comparing the implicit and explicit method, the increased computational cost for the implicit method is rebalanced by significantly improved numerical stability of the algorithm. This allows for solving the ODE with fewer steps, as deeper investigated in Sec. 6.3.

Based on the original idea of the Euler methods, a variety of numerical solvers can be derived, but are not key focus here and therefore not further investigated. The explicit as well as the implicit method are part of the group of *single-step solvers*, that predict the next state based on a *single* given step. To provide a common interface, often (e.g. in Chen *et al.* [16]) a wildcard algorithm $ODESolve$ is defined to solve an ODE from t_0 to t_f, like

$$\boldsymbol{x}_f = ODESolve(\boldsymbol{f}, \boldsymbol{x}_0, \boldsymbol{p}, t_0, t_f), \tag{2.10}$$

which captures all single-step methods without investigation of the actual algorithmic structure, including if the method is explicit or implicit. Because more advanced solvers in general feature *dynamic step size control*, meaning the step size h is automatically determined by stability and tolerance requirements, the step size is not input to $ODESolve$ and the number of steps to reach $\boldsymbol{x}_f$ starting at $\boldsymbol{x}_0$ is not investigated within this algorithm definition. If more than just the terminal state $\boldsymbol{x}_f$ is required, so a *state trajectory* $\boldsymbol{X} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_f\}$, the function $ODESolve$ can be called multiple times for consecutive time segments, which we refer to as $ODESim$.

Event ordinary differential equations
Further, ODEs can feature *events*. Events can be seen as special time instances that depend on the system state and/or time. At an event, the system state is allowed but not required to change discontinuously[1]. If events solely depend on time itself, e.g. a chemical component is added to another, they are referred to as *time events*. If events depend on the system state, e.g. a ball hits the ground, they are called *state events*. In engineering practice, one specific event often depends either on time or states, and in general not on both.

Introducing events results in a different interface for the ODE solver, because the termination time now depends on the solution trajectory and is not known beforehand [16]. Single-step solvers that provide event-handling can therefore be expressed by an extended algorithm interface:

$$t^*, \boldsymbol{x}(t^*) = ODESolveEvent(\boldsymbol{f}, \boldsymbol{x}_0, \boldsymbol{p}, t_0, t_f, c_e). \tag{2.11}$$

The terminal time t_f is still an algorithm argument, but now defines only an *upper bound* for the simulation time span. We find a second output of the algorithm t^*, determining the *actual* simulation stop time. This means that the solver — instead of the user — decides on how far the simulation progresses and terminates as soon as a new event at t^* is found. For the case that no event happens within $[t_0, t_f]$, it states that $t^* = t_f$. To determine events, the event function $c_e(\boldsymbol{x}(t), \boldsymbol{p}, t)$ is introduced. For any time instance that features an event t^*, it states that the event function is zero, so

$$c_e(\boldsymbol{x}(t^*), \boldsymbol{p}, t^*) = 0. \tag{2.12}$$

Later we will introduce how such an event function is defined in practice using *event indicators*. After termination of $ODESolveEvent$, the state can be updated discontinuously (if required) and a new integration can be started, calling $ODESolveEvent$ again — until we reach the target termination time for the simulation, so $t^* = t_f$. The subsequent calls to $ODESolveEvent$ until reaching t_f can be summarized again in the abstract algorithm $ODESimEvent$.

Partial differential equation (PDE)
PDEs contain not only the derivative with respect to a single *independent* variable — in case of ODEs often time itself — but multiple variables. This directly leads to the investigation of partial instead of total derivatives, because multiple variables to differentiate along are available. Therefore, we find that an ODE is a simplified case of the generic PDE. Contrary to ODEs, we can not solve PDEs by directly applying a time stepping solver (referred to as *discretization over time*), but need to perform an additional step of *spatial discretization*. Within this spatial discretization step, we decide how to handle the partial derivatives within the equations. A variety of methods for this step exist, perhaps the easiest one is to apply finite differences as introduced in Sec. 2.1.3.2.

Whereas PDEs play a significant role in engineering, especially in fluid dynamics and heat transfer, in this work we focus only on PDEs after spatial discretization — mathematically resulting in ODEs.

[1]The discontinuous change does not need to be within the continuous state, but can also be within the discrete part of the system.

Differential algebraic equation (DAE)
The same as for ODEs, DAEs also describe the change of quantities $\boldsymbol{x}$ over time by defining an equation depending on $\dot{\boldsymbol{x}}$. However, the general form of the DAE is different compared to the one of the ODE:

$$\mathbf{0} = \boldsymbol{F}(\dot{\boldsymbol{x}}(t), \boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t). \tag{2.13}$$

The core difference is indeed, that the function $\boldsymbol{f}$ for ODEs is solved for $\dot{\boldsymbol{x}}$, whereas for DAEs the differentiated state $\dot{\boldsymbol{x}}$ is explicitly part of the right-hand side $\boldsymbol{F}$. As a consequence, any ODE can be transformed into the notation of a DAE by subtracting $\dot{\boldsymbol{x}}$ from both sides of the equation — in fact any ODE is a DAE. However, not any DAE can be transformed into an ODE. By applying the implicit function theorem, we find that solving an DAE for $\dot{\boldsymbol{x}}$ is only possible if the Jacobian $\partial \boldsymbol{F}/\partial \dot{\boldsymbol{x}}$ is not singular. If the Jacobian is singular, methods for index reduction such as Pantelides algorithm[2] [17] can be applied for trying to reduce the DAE index to obtain an ODE. Whereas many real world DAEs can be reduced to ODEs, the success of index reduction is not always guaranteed. Many modeling approaches (like object-oriented modeling) lead to DAEs, however in this work we assume that in this case index reduction is applied, and we can proceed to further work with an ODE.

2.1.3 Sensitivity analysis (SA) for ODE solutions

One of the core concepts and backbone of most deep learning applications is local sensitivity analysis (SA). In addition to local SA, there is also the subject of global SA, which deals with the influence of variable *ranges* or distributions. However, in this work, we only investigate local SA and omit the adjective "local". SA describes a class of methods, that allows for determination of sensitivities between at least two variables. Essentially, the sensitivity describes how one variable changes in (causal) relation to another variable. The term *sensitivity* refers not only to scalar derivatives, but also to gradients (vectors) and Jacobians (matrices), as well as higher-order derivatives like Hessians. Sensitivities are used in a very wide range of applications and are a core tool for model parameterization and validation in engineering practice.

In ML, the term *training* of models refers to the optimization of parameters with respect to an objective function. In supervised learning, the objective function is often minimized and referred to as *loss function* or *loss* in short. Opposite, in reinforcement learning it is common to maximize the objective and refer to it as *reward*. The problem of optimization is solved by algorithms called *optimizers*. Basically, optimizers can be subdivided into *gradient-free* and *gradient-based* methods. Because of the very large parameter dimension in ML, the chance of decreasing the loss value by random or heuristic perturbation of parameters decreases and the importance of a systematic optimization in direction of a decreasing loss value becomes necessary. As a result, in general gradient-based optimizers are applied here. The most popular gradient-based optimizers are *gradient descent* algorithms, that are compared in Ruder [18]. In the following, we investigate determination of a loss function gradient, that involves solving a parameterizable ODE, in order to perform gradient-based optimization.

[2]Interestingly, the algorithm was originally invented for initialization of DAEs instead of for index reduction.

To investigate SA for ODE solutions, we define the continuous loss function as

$$l_{con}(\boldsymbol{x}(t), \boldsymbol{p}) = \int_{t_0}^{t_f} h(\boldsymbol{x}(t), \boldsymbol{p}) dt, \tag{2.14}$$

where $\boldsymbol{x}(t)$ is the ODE solution, evaluated at t. Here, h is the function that rates the deviation of a given state, which again is integrated to the final loss value. However, the continuous approach requires a continuous ODE solution, which is not practicable because ODEs, as already introduced, are usually solved for discrete points in time. At the latest if h compares the state to *real data*, the discrete approach must be adopted, as computer measurements are always (at least temporally) discretized. The time-discretized version of the loss for a solution $\boldsymbol{X} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_n\}$ with $n+1$ points states [19]:

$$l(\boldsymbol{X}, \boldsymbol{p}) = \sum_{i=0}^{n} h(\boldsymbol{x}_i, \boldsymbol{p}). \tag{2.15}$$

Note, that proper temporal discretization of the continuous loss l_{con} would require multiplication with the duration of the individual time slices $\frac{t_f - t_0}{n}$, if we assume equidistant time samples. This is usually omitted in practice. However, weighting factors can be applied to the individual points if required (e.g. by designing the function h).

Furthermore, we want the loss function to depend only on $\boldsymbol{p}$, and not on $\boldsymbol{X}$ — indeed we know that the discretized solution $\boldsymbol{X}$ depends on $\boldsymbol{p}$ if we solve the ODE. We can achieve this by inserting $ODESim$ in the loss function:

$$l(\boldsymbol{p}) \stackrel{\text{def}}{=} l(\boldsymbol{X} = ODESim(\boldsymbol{p}), \boldsymbol{p}), \tag{2.16}$$

where

$$ODESim(\boldsymbol{p}) \stackrel{\text{def}}{=} ODESim(\boldsymbol{f}, \boldsymbol{x}_0, \boldsymbol{p}, t_0, t_f). \tag{2.17}$$

As stated, gradient-based optimizers require the gradient, so the derivative of the loss function w.r.t. the optimization parameters. Applying the chain rule, we obtain:

$$\frac{dl(\boldsymbol{p})}{d\boldsymbol{p}} = \frac{dl(\boldsymbol{X} = ODESim(\boldsymbol{p}), \boldsymbol{p})}{d\boldsymbol{p}} \tag{2.18}$$

$$= \frac{\partial l(\boldsymbol{X}, \boldsymbol{p})}{\partial(\boldsymbol{X})} \frac{d(\boldsymbol{X} = ODESim(\boldsymbol{p}))}{d\boldsymbol{p}} + \frac{\partial l(\boldsymbol{X}, \boldsymbol{p})}{\partial \boldsymbol{p}}. \tag{2.19}$$

For the sake of readability, we omit the function arguments in the following and write

$$\frac{dl}{d\boldsymbol{p}} = \frac{\partial l}{\partial \boldsymbol{X}} \frac{d\boldsymbol{X}}{d\boldsymbol{p}} + \frac{\partial l}{\partial \boldsymbol{p}}. \tag{2.20}$$

Sometimes, the loss function is not directly dependent on the parameters — the loss function compares the ODE solution to some reference — so the gradient simplifies to

$$\frac{dl}{d\boldsymbol{p}} = \frac{\partial l}{\partial \boldsymbol{X}} \frac{d\boldsymbol{X}}{d\boldsymbol{p}}. \tag{2.21}$$

However, regularization terms often depend directly on $\boldsymbol{p}$ (think of L1 or L2 regularization of layer weights), which motivates the investigation of Eq. 2.20 instead of Eq. 2.21 for the universal case [20].

For the computation of $^{dl}/_{d\boldsymbol{p}}$, we can apply different methods of SA. Whereas there are multiple well-established algorithms for sensitivity computation and estimation, the special challenge for the loss function gradient within this work is that it contains the solution of an ODE, which requires *solving* the ODE because the solution is sensitive w.r.t. parameters. Sapienza *et al.* [20] offers a valuable classification of the available methods for this purpose, as can be seen in Fig. 2.1. However, additional modifications are necessary to apply the introduced methods to *event ODEs*, as they appear in engineering practice as well. Sensitivity analysis of ODEs under different kinds of discontinuities is nicely discussed in Pfeiffer [21]. However, we do not deal with this subject in full depth, as it is not relevant to follow the core contributions of this work.

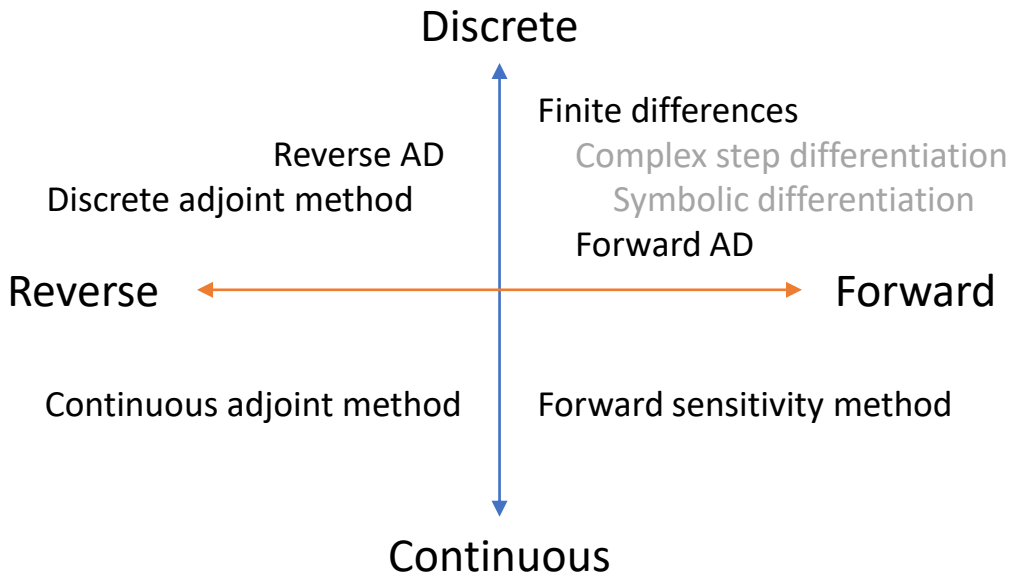


Figure 2.1: Available methods for SA of ODE solutions, classified regarding the properties discrete/continuous and reverse/forward. This figure is a slightly modified version of Sapienza *et al.* [20]. Methods, that are not further investigated within this thesis are gray-colored.

The figure classifies available methods regarding two properties:

Continuous/Discrete Sensitivity estimation over an ODE finally results into numerical values for the sensitivities and therefore necessarily involves two steps: First, the solving of the ODE, which is also referred to *discretization* because the solution of the ODE is discretized over time. Second, the actual *differentiation* to obtain the partial derivatives that are necessary to evaluate the total loss function gradient. The investigated methods can therefore be classified regarding the fact that discretization happens first, which refers to *discrete* methods [20], sometimes also referred to as *discretize-then-differentiate* [22] or *discretize-then-optimize* [23]. In contrast, if differentiation happens first, we can refer to *continuous* methods [20], sometimes also referred to as *differentiate-then-discretize* [22] or *optimize-then-discretize* [23]. As a final note, classification is not always straightforward and algorithms exist, that operate in between these two extrema.

Forward/Reverse Based on the two steps for sensitivity determination over ODE solutions introduced in the previous paragraph, a further distinction can be made in addition: If the sensitivity computation happens in the same direction as the solution process (so in temporal direction of the ODE solution), a method is referred to as *forward*, whereas it is called *reverse* in the opposite case. With focus on a loss function, forward methods determine sensitivities with respect to the loss function input (here: the parameters), therefore compute the partial derivatives of intermediate results w.r.t. the input — and finally the output w.r.t. the input. Reverse methods on the other hand, compute the partial derivatives of the output (the loss function value) w.r.t. intermediate results — and again, finally the output w.r.t. the input.

The most relevant methods for SA within this thesis are briefly introduced in the upcoming subsections.

2.1.3.1 Automatic differentiation (AD)

Automatic differentiation (AD) is a *discrete* approach for determination of sensitivities, and can be applied as long as all operations in between the involved variables are differentiable. This is possible without any further arrangements if a language is used that allows for differentiable programming, such as *Julia* (s. Sec. 2.3.2).

Because the right-hand side of the ODE, as well as the ODE solver consist of differentiable operations like additions and multiplications — or in general: operations with known derivatives — it is possible to provide an implementation that can be examined via AD. AD provides a significant advantage compared to continuous approaches: Event ODEs and mixed continuous-discrete systems in general, involving functions for locating events (*event indicators*) and handling (*event affect*), can easily be examined using AD as long as all operations are implemented differentiable. However, especially for challenging system properties, further measures need to be implemented in order to provide numerical stability. Other approaches, as introduced in the following subsections, require the extension of the introduced algorithms for SA in order to support events.

AD is available in two different operation directions: forward and reverse. The two operation modes are briefly introduced at the following example equation:

$$y(x_1, x_2) = x_1^2 \cdot sin(x_2). \tag{2.22}$$

For this equation, we want to determine the partial derivatives $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$. We subdivide the equation into *primal* equations, from which we know the derivatives analytically, and introduce new intermediate variables ω_i to prevent overwriting

existing values:

$$\omega_1 = x_1 \Rightarrow \frac{d\omega_1}{dx_1} = 1 \tag{2.23}$$

$$\omega_2 = x_2 \Rightarrow \frac{d\omega_2}{dx_2} = 1 \tag{2.24}$$

$$\omega_3 = \omega_1^2 \Rightarrow \frac{d\omega_3}{d\omega_1} = 2\omega_1 \tag{2.25}$$

$$\omega_4 = sin(\omega_2) \Rightarrow \frac{d\omega_4}{d\omega_2} = cos(\omega_2) \tag{2.26}$$

$$\omega_5 = \omega_4 \cdot \omega_3 \Rightarrow \frac{d\omega_5}{d\omega_4} = \omega_3 \quad \text{and} \quad \frac{d\omega_5}{d\omega_3} = \omega_4. \tag{2.27}$$

Because every variable ω_i with $i \in \{1, ..., 5\}$ is used exactly once, we can easily visualize the corresponding computation as a *computational graph* (compare Fig. 2.2 and 2.3) for the following observations.

To have results to compare the two approaches with, we apply the chain rule and obtain the analytical derivatives:

$$\frac{\partial y}{\partial x_1} = \frac{\partial \omega_5}{\partial \omega_3}\frac{\partial \omega_3}{\partial \omega_1}\frac{\partial \omega_1}{\partial x_1} = \omega_4 \cdot 2\omega_1 \cdot 1 = \omega_4 \cdot 2\omega_1 \quad \text{and} \tag{2.28}$$

$$\frac{\partial y}{\partial x_2} = \frac{\partial \omega_5}{\partial \omega_4}\frac{\partial \omega_4}{\partial \omega_2}\frac{\partial \omega_2}{\partial x_2} = \omega_3 \cdot cos(\omega_2) \cdot 1 = \omega_3 \cdot cos(\omega_2). \tag{2.29}$$

The major difference between forward and reverse AD from a mathematical perspective is the evaluation order of this chain, in forward AD it is perturbed with a seed vector and evaluated from right to left, leading to a series of Jacobian-vector-products (JVPs). In reverse AD on the other hand, the evaluation is performed from left to right, leading to a series of vector-Jacobian-products (VJPs). Both are investigated in detail in the following.

Forward-mode

As stated, in forward-mode AD the derivative chain is evaluated from right to left, or from inside to outside if nested functions are investigated. This mirrors the natural computational order and allows for a very intuitive determination of derivatives: Any time a primal computation happens, the same computation on derivative level can be applied, because all required variable values are available (known), compare Fig. 2.2. However, it is important to clarify, that even if the partial derivative operations are known symbolically, the partial derivatives are evaluated at every computational step and intermediate results are stored numerically.

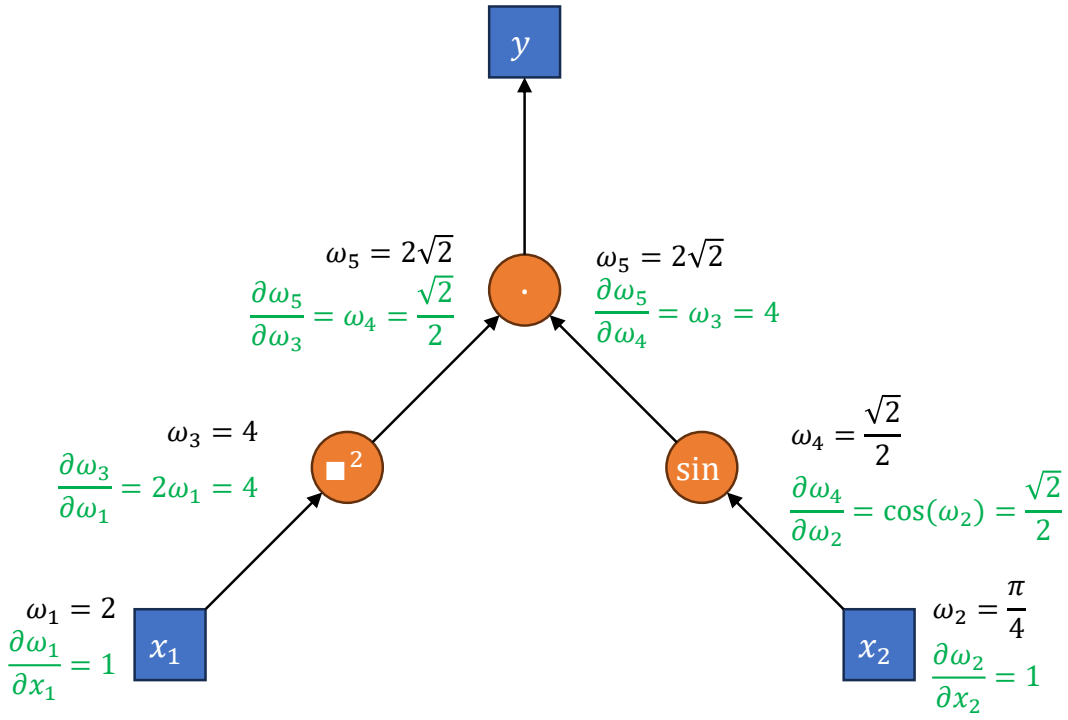


Figure 2.2: The computational graph for the example function for forward AD. The graph is evaluated starting with x_1 (for the derivative $\partial y/\partial x_1$) or x_2 (for the derivative $\partial y/\partial x_2$). The derivatives (green) are computed side-by-side with the actual function values (black), preserving the *natural* evaluation order.

Strictly speaking, forward-mode AD constructs a function s_{fw}, which computes a partial derivative for a given seed, meaning:

$$s_{fw}\left(\frac{\partial \omega_1}{\partial x_1}, \frac{\partial \omega_2}{\partial x_2}\right) = \frac{\partial \omega_1}{\partial x_1}\frac{\partial \omega_3}{\partial \omega_1}\frac{\partial \omega_5}{\partial \omega_3} + \frac{\partial \omega_2}{\partial x_2}\frac{\partial \omega_4}{\partial \omega_2}\frac{\partial \omega_5}{\partial \omega_4}. \tag{2.30}$$

Here, $\frac{\partial \omega_1}{\partial x_1}$ and $\frac{\partial \omega_2}{\partial x_2}$ are referred to as *seeds*. For proper evaluation of the partial derivatives it is necessary to provide a one-hot[3] seed vector. This means for n partial derivatives, n evaluations are required. For our example, this results in two evaluations of the sensitivity function:

$$\frac{\partial y}{\partial x_1} = s_{fw}(1, 0) \quad \text{and} \tag{2.31}$$

$$\frac{\partial y}{\partial x_2} = s_{fw}(0, 1). \tag{2.32}$$

A single inference of the computational graph is referred to as *sweep* — meaning two sweeps are required to solve SA via forward AD here.

Reverse-mode

Because we need to infer the computational graph starting at the leaves (so x_1 and x_2), forward AD scales poorly with increasing parameter count, because the number of leaves increases. On the other hand, the number of roots (here y) always stays

[3]Exactly one seed is 1, all others are 0.

1, as long as we investigate single-valued loss functions. This observation motivates the inference of the computational graph *in reverse* for AD, so starting at the root and propagating computations to the leaves. However, this introduces additional challenges, as explained in the following.

For the sake of readability, we introduce the variable

$$\bar{\omega}_i = \frac{\partial y}{\partial \omega_i}, \tag{2.33}$$

which is called *adjoint* (not to be confused with the continuous/discrete adjoint method) or cotangent. Note, that the adjoint describes the sensitivity w.r.t. the root of the computational tree (here y).

Beneficial, the reverse AD sensitivity function $\boldsymbol{s}_{rv}$ is able to compute all sensitivities with only a single evaluation (*sweep*) of the computational tree:

$$\boldsymbol{s}_{rv}(\bar{\omega}_5) = \bar{\omega}_5 \cdot \begin{bmatrix} \frac{\partial \omega_5}{\partial \omega_3}\frac{\partial \omega_3}{\partial \omega_1}\frac{\partial \omega_1}{\partial x_1} \\ \frac{\partial \omega_5}{\partial \omega_4}\frac{\partial \omega_4}{\partial \omega_2}\frac{\partial \omega_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \bar{\omega}_3 \frac{\partial \omega_3}{\partial \omega_1}\frac{\partial \omega_1}{\partial x_1} \\ \bar{\omega}_4 \frac{\partial \omega_4}{\partial \omega_2}\frac{\partial \omega_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \bar{\omega}_1 \frac{\partial \omega_1}{\partial x_1} \\ \bar{\omega}_2 \frac{\partial \omega_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix}. \tag{2.34}$$

For the reverse AD case, $\bar{\omega}_5 = 1$ is referred to as the *seed*.

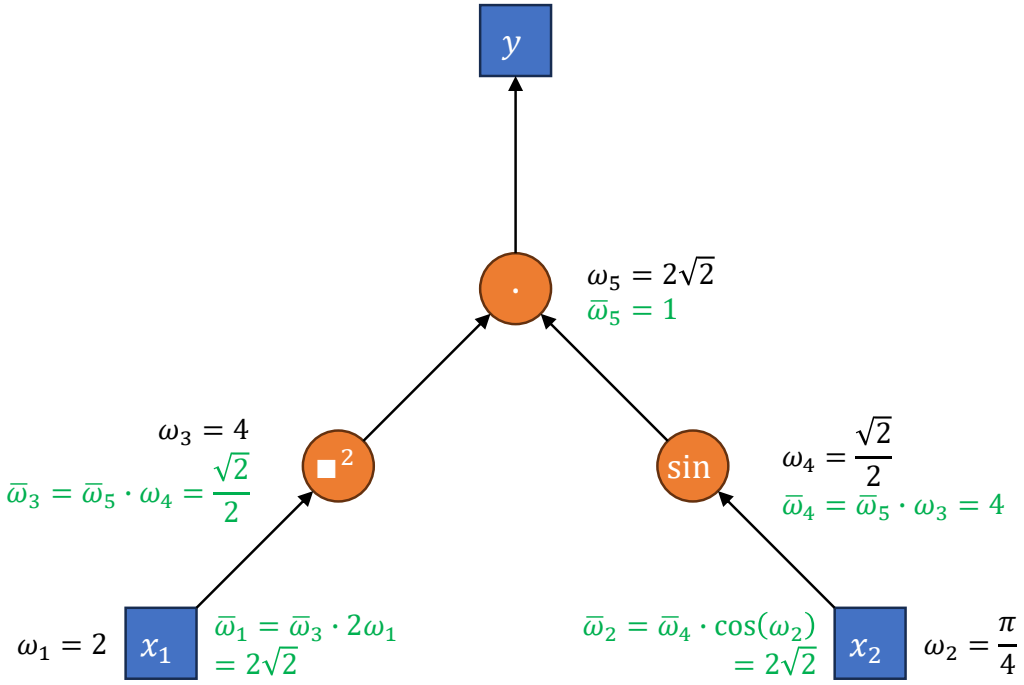


Figure 2.3: The computational graph for the reverse AD pass. To determine the sensitivities $\bar{x}_1$ and $\bar{x}_2$, the computational graph is evaluated a single time from top to bottom. However, the intermediate values of ω_i, that are needed to compute the partial derivatives, need to be computed beforehand by evaluation of the computational graph in *forward* direction and storing the results.

For the general case that we differentiate a scalar cost function, the computational graph only needs to be evaluated a single time to determine all partial derivatives. This results in a total cost of $\mathcal{O}(k + |\boldsymbol{p}|)$ [20], where k is the number of VJP operations. However, all intermediate variables need to be determined

(this can be performed during the forward evaluation of the graph to determine the value of y) and saved. Whereas this requirement seems trivial for the current example, storing these intermediate variables can be inefficient or not feasible at all for large computational graphs. Note, that the number of VJPs scales with the number of steps, that the ODE solver performs during solving. In cases with large computational graphs, values can be recomputed, or *checkpointing* can be applied, that features a mix of stored and recomputed variables [20]. Another obvious challenge are branches in the computational graph, that can be tackled by a variety of methodological adjustments. Because these adjustments always come with a downside (often the computational cost), multiple AD libraries exist, targeting on different application requirements and problem sizes.

To summarize, for the general case of a vectorial loss function, $|\boldsymbol{l}|$ evaluations of the graph are required, resulting in a cost of $\mathcal{O}(k \cdot |\boldsymbol{l}| + |\boldsymbol{p}|)$ [20] for reverse AD. The general cost for forward AD is $\mathcal{O}(k \cdot |\boldsymbol{p}| + |\boldsymbol{l}|)$ [20], which clearly motivates the use of reverse mode AD for gradient-based training in ML based on a scalar loss function[4], so $|\boldsymbol{p}| >> |\boldsymbol{l}| = 1$. For artificial neural networks (ANNs), reverse AD is often referred to as *backpropagation*.

2.1.3.2 Finite differences

When applying *finite differences*, the partial derivatives are approximated with a finite differences scheme. For example, the first order forward differences scheme states

$$\frac{\partial f(p)}{\partial p} = \frac{f(p + \Delta p) - f(p)}{\Delta p} + \mathcal{O}(\Delta p), \tag{2.35}$$

whereas the central differences scheme states

$$\frac{\partial f(p)}{\partial p} = \frac{f(p + \Delta p) - f(p - \Delta p)}{2\Delta p} + \mathcal{O}(\Delta p^2). \tag{2.36}$$

We notice, that applying finite differences introduces a *truncation error*, in case of forward differences this is $\mathcal{O}(\Delta p)$, for central differences it is $\mathcal{O}(\Delta p^2)$. Whereas central differences features a smaller truncation error (for small $\Delta p < 1$), it requires two function evaluations instead of one, if we assume that the evaluation $f(p)$ is computational free because we perform it for solving the ODE anyway. To conclude, independent of the chosen finite differences scheme, the choice of Δp is crucial, because large values induce a large truncation error, but small values lead to floating point issues [24].

Interestingly, by replacing p with time t and f with x, and solving the forward differences for $x(t + \Delta t)$, we find the explicit Euler method for solving ODEs, which emphasizes the strong relationship between PDEs and ODEs once again. In general, finite differences are computational expensive, e.g. for forward differences we need to perform n evaluations ($2n$ for central differences) for $n = |\boldsymbol{p}|$ parameters. In the case of SA for an ODE solution, this results in performing the simulation of the system n (or $2n$ respectively) times. Applying this method to problems of the parameter dimension of ML, often featuring millions[5] of parameters, is not

[4]However, reverse AD is of course still efficient if multi-valued loss functions are used as long the loss dimension is small compared to the parameter dimension.

[5]There are even models with *trillions* of parameters, e.g. *BaGuaLu* features 14.5 trillion parameters, opening up to 174 trillion parameters [25].

feasible. Even within hybrid modeling applications, which operate in the order of thousands, this is at least inefficient. While for Jacobians there are methods for sparse sampling[6] (like Jacobian coloring) that can be applied to significantly reduce the computational effort, such approaches are not beneficial for the determination of (loss function) gradients.

Interestingly, if *nested AD*, so partial derivatives of other partial derivatives, is required (like for example for Eigen-informed regularization as introduced in Sec. 6.3), other methods of SA, as we will introduce in the following, become computationally inefficient and hard to implement. In such cases, pairing an efficient method of SA for the determination of one partial derivative and finite differences for the other is a common approach to achieve a satisfactory computing performance.

Events

Finally, the same as for AD, finite differences directly allow us to investigate event ODEs, because by permutation of parameters proper operation of event-handling is not affected. However, the choice of a suitable step size is very challenging, since even small steps can have a very strong influence on the solution and/or already lead to entering areas of the state space that do not lead to a solvable system (e.g. unsolvable algebraic loops or modeled assertions, see Sec. 6.5). Values that are too small can lead to numerical difficulties. And finally, challenging systems may even require adapting the sampling step size during solving and across different states, making finite differences not only a computationally expensive measure, but also one that presents further challenges.

2.1.3.3 Forward sensitivity method

The forward sensitivity method is part of continuous SA for ODE solutions, which is summarized in Hindmarsh *et al.* [26]. In the following, the sensitivity of the system state w.r.t. the i-th parameter is abbreviated with

$$\boldsymbol{s}_i(t) = \frac{\partial \boldsymbol{x}(t)}{\partial p_i}, \tag{2.37}$$

and the Jacobian of the right-hand side regarding the system state as

$$\boldsymbol{A}(t) = \frac{\partial \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{p}, t)}{\partial \boldsymbol{x}(t)} \stackrel{\text{def}}{=} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}. \tag{2.38}$$

The matrix $\boldsymbol{A}$ is often referred to as *system matrix* in systems and control theory. The forward sensitivity method is based on the observation, that the total derivative w.r.t. time can also be defined for sensitivities, in this case

$$\dot{\boldsymbol{s}}_i(t) = \frac{d\boldsymbol{s}_i(t)}{dt} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{x}}{\partial p_i} + \frac{\partial \boldsymbol{f}}{\partial p_i} = \boldsymbol{A}\boldsymbol{s}_i + \frac{\partial \boldsymbol{f}}{\partial p_i}. \tag{2.39}$$

Inserting Eq. 2.37 and Eq. 2.38, we find the applied chain rule [27]. Solving requires the determination of the involved partial derivatives and augmentation of the system state vector by the new time derivatives for the parameter sensitivities.

[6] However, even if sparse systems are investigated, sparsity information is not always available and not trivial to determine for event systems.

In practice, the three partial derivatives on the right-hand side of Eq. 2.39 are not determined individually. The first summand, the JVP can be computed through AD or approximated at once, for example via central differences [26]:

$$\boldsymbol{A}\boldsymbol{s}_i \approx \frac{\boldsymbol{f}(\boldsymbol{x} + \sigma \boldsymbol{s}_i, \boldsymbol{p}, t) - \boldsymbol{f}(\boldsymbol{x} - \sigma \boldsymbol{s}_i, \boldsymbol{p}, t)}{2\sigma}, \tag{2.40}$$

with step size σ. Beneficial, this way the error control within the solver[7] also keeps track of the error of the approximated sensitivities [26]. However, this method has a high cost of $\mathcal{O}(|\boldsymbol{x}| \cdot |\boldsymbol{p}|)$ [19], making the forward sensitivity method only attractive for small ODEs. Due to the limited applicability for large systems, applicability for event ODEs is not further investigated for this method.

2.1.3.4 Continuous adjoint method

The poor computational scaling of the forward sensitivity method motivates the use of other methods for sensitivity analysis if $|\boldsymbol{p}|$ is large. The (continuous) adjoint method fulfills this requirement by providing a cost of $\mathcal{O}(|\boldsymbol{x}| + |\boldsymbol{p}|)$ [19]. In contrast to the forward sensitivity method, which computes many partial derivatives $\partial \boldsymbol{x}/\partial p_i$ for $i \in \{1, ..., |\boldsymbol{p}|\}$, the adjoint method directly computes the gradient of the cost function, or a state-dependent function in general, w.r.t. *all* parameters $dl/d\boldsymbol{p}$ at once. This gradient can be used directly for solving an optimization task.

Based on the loss function in Eq. 2.15, the adjoint method, originally introduced in [28], divides into three steps: **First**, the original ODE is solved to obtain the solution $\boldsymbol{x}(t)$ of the ODE. **Second**, the *adjoint problem* is solved backwards in time. The general adjoint problem states [27]:

$$\dot{\boldsymbol{\lambda}}^T(t) = -\boldsymbol{\lambda}^T(t)\boldsymbol{A}(t) - \frac{\partial h(\boldsymbol{x}(t))}{\partial \boldsymbol{x}(t)} \quad \text{with} \quad \boldsymbol{\lambda}(t_f) = \boldsymbol{0}. \tag{2.41}$$

In literature, often the notation $\boldsymbol{\lambda}'$ is found instead of $\boldsymbol{\lambda}^T$. This is because in the space of real numbers, the adjoint operation (conjugate transpose) reduces to a simple transpose operation.

Here, it is important to note that the dimension of the adjoint is determined by the size of the matrix $\boldsymbol{A}$ and therefore the number of *states*. The size of the ODE during forward sensitivity analysis, on the other hand, depends on the number of *parameters*.

Third, the actual loss function gradient can be computed by solving the integral [20], [26], [27]:

$$\frac{dl}{d\boldsymbol{p}} = \boldsymbol{\lambda}^T(t_0)\frac{\partial \boldsymbol{x}(t_0)}{\partial \boldsymbol{p}} + \int_{t_0}^{t_f} (\boldsymbol{\lambda}^T \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} + \frac{\partial h}{\partial \boldsymbol{p}}) dt. \tag{2.42}$$

This shows very clearly that although two additional ODEs must be solved for the continuous adjoint method, it is less expensive to solve two small ODEs ($|\boldsymbol{x}|$ and $|\boldsymbol{p}|$) instead of a single large one ($|\boldsymbol{p}| \cdot |\boldsymbol{x}|$). As already stated, this approach is also referred to as *differentiate-then-discretize*, because the discretization of the sensitivities (w.r.t. time) happens in the very last step, during solving the integral. This integral can be computed in different ways, however the VJP within can not be easily sampled like the JVP within the forward sensitivity method. Here, reverse

[7] Of course only if we use a solver with step size control.

AD (as introduced in Sec. 2.1.3.1) offers an efficient way for determination of this VJP[8]. Indeed, the continuous adjoint method is often paired with AD in practice.

To summarize, the continuous adjoint method computes the required sensitivities by solving the adjoint ODEs with the size of the *system state*, independent of the parameter count. Only the final integral features the size of the parameter space, but can be evaluated efficiently based on the intermediate results of $\boldsymbol{x}(t)$ and $\boldsymbol{\lambda}(t)$.

Alternative representation

Further, some publications [11], [16], [19] apply a different version of the continuous adjoint method, that only rates the final state of the integration $\boldsymbol{x}_f = \boldsymbol{x}(t_f)$ as part of the loss function $l_f(\boldsymbol{x}_f, \boldsymbol{p})$, and no intermediate solution points. This simplified adjoint equation (backwards in time) is defined as

$$\dot{\boldsymbol{\lambda}}^T(t) = -\boldsymbol{\lambda}^T(t)\boldsymbol{A}(t) \quad \text{with} \quad \boldsymbol{\lambda}(t_f) = \left(\frac{\partial l_f(\boldsymbol{x}(t_f), \boldsymbol{p})}{\partial \boldsymbol{x}(t_f)}\right)^T, \tag{2.43}$$

and the final loss function gradient is calculated as

$$\frac{dl_f}{d\boldsymbol{p}} = -\int_{t_1}^{t_0} \boldsymbol{\lambda}(t)^T \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{p}} dt. \tag{2.44}$$

Compared to Eq. 2.42, this integral is solved backward in time. Because we rate intermediate points as well within this work, the alternative representation is only mentioned for completeness and not investigated further.

Events

In the following, we briefly investigate how the continuous adjoint method can be applied to event ODEs as well. We observe, that the solution of an event ODE consists of continuous (in terms of *continuous differentiable*) sections, that are connected by discontinuous *jumps* in the trajectory. At the discontinuous locations — so at *events* — the system is in general not differentiable. We summarize, that the solution consists of multiple continuous functions with a priori unknown[9] stop time, and can be obtained by multiple consecutive evaluations of $ODESolveEvent$ (compare Eq. 2.11).

For investigation of the differentiability of this approach, Chen *et al.* [16] offers a valuable point of view. In the following, we briefly summarize the key insights of the underlying derivation. $ODESolveEvent$ is separated into two parts: First, the root of the event function is determined and we obtain t^*. Second, we can use $ODESolve$ to solve the actual ODE from t_0 to t^*, because we know that within this segment the solution is continuous. This allows us to reuse the already derived continuous adjoint method for this second step and only requires mathematically deriving sensitivities for the root search procedure. A more detailed investigation is part of Chen *et al.* [16] and will not be discussed further here, but is also not relevant for understanding the methods presented in this thesis.

[8]Remark for Julia developers: This corresponds to the VJP choice (keyword) when working with differentiate-then-discretize methods.

[9]The stop time depends on the state (so the ODE solution) and the actual event (termination) time is therefore not known beforehand.

2.1.3.5 Discrete adjoint method

For the discrete version of the adjoint sensitivity method, sensitivities are determined based on the temporal *discretized* ODE solution. Every step of the solving process (so every step of the numerical solver) is reformulated as nonlinear system of equations, referred to as *constraint* [20], resulting in a constraint $\boldsymbol{g}_i$ for every solver step $i \in \{1, ..., n\}$:

$$\boldsymbol{G}(\boldsymbol{X}, \boldsymbol{p}) = (\boldsymbol{g}_1(\boldsymbol{X}, \boldsymbol{p}), ..., \boldsymbol{g}_n(\boldsymbol{X}, \boldsymbol{p})) = \boldsymbol{0}. \tag{2.45}$$

Note, that we omitted the trivial constraint for the initial state $\boldsymbol{x}_0$. At the example of the backward Euler method, such a constraint may be constructed by bringing all operations to one side:

$$\boldsymbol{g}_{i+1}(\boldsymbol{x}_{i+1}, \boldsymbol{x}_i, \boldsymbol{p}) = \boldsymbol{x}_i + h \cdot \boldsymbol{f}(\boldsymbol{x}_{i+1}, \boldsymbol{p}, t_{i+1}) - \boldsymbol{x}_{i+1} = \boldsymbol{0} \quad \text{for} \quad i \in \{0, ..., n-1\}. \tag{2.46}$$

We note that $\boldsymbol{x}_i$ is known, because for $i = 0$ the initial state $\boldsymbol{x}_0$ is given, and $\boldsymbol{x}_i$ is calculated in the i-th step for $i > 0$ — so the nonlinear system of equations is solved for $\boldsymbol{x}_{i+1}$ for every step. Even if we are interested in finding a valid solution for a given ODE, which is true for $\boldsymbol{G} = \boldsymbol{0}$, we can evaluate $\boldsymbol{G}$ for any given $\boldsymbol{X}$ and obtain the *residual* — which opens up new opportunities.

Recalling the loss sensitivity equation in Eq. 2.20, we find that it is inefficient to compute the Jacobian $\partial \boldsymbol{X} / \partial \boldsymbol{p}$ for large numbers of states and parameters, because this requires differentiation over every solver step, involving solving a nonlinear system of equations. We further notice that, similar to other approaches introduced, we are actually interested in the result of the matrix product

$$\frac{\partial l}{\partial \boldsymbol{X}} \frac{\partial \boldsymbol{X}}{\partial \boldsymbol{p}} \tag{2.47}$$

instead of $\partial \boldsymbol{X} / \partial \boldsymbol{p}$ directly. The core idea can therefore be summarized as follows: Changes in the parameter gradient can not only be tracked by monitoring the ODE solution directly, but also by monitoring the deviation in the constraint. As a consequence, instead of directly differentiating the loss function $\boldsymbol{l}$, the constraint can be differentiated [20]:

$$\frac{d\boldsymbol{G}}{d\boldsymbol{p}} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{X}} \frac{\partial \boldsymbol{X}}{\partial \boldsymbol{p}} = \boldsymbol{0}. \tag{2.48}$$

With the goal to substitute the expensive Jacobian $\partial \boldsymbol{X} / \partial \boldsymbol{p}$, we can solve the middle expression of Eq. 2.48 for $\partial \boldsymbol{X} / \partial \boldsymbol{p}$ and insert into the loss sensitivity equation Eq. 2.20, that now states [20]:

$$\frac{dl}{d\boldsymbol{p}} = \frac{\partial l}{\partial \boldsymbol{p}} - \frac{\partial l}{\partial \boldsymbol{X}} \left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{X}} \right)^{-1} \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}. \tag{2.49}$$

Note, that this equation does indeed not require differentiation of $\boldsymbol{X} = ODESim(\boldsymbol{p})$ anymore, but now the constraint $\boldsymbol{G}(\boldsymbol{X}, \boldsymbol{p})$, that only depends on the already calculated solution $\boldsymbol{X}$. Finally, we only need to connect between the sensitivities of the constraint and the loss function (that we are actually interested in) by solving for the *adjoint* defined as [20]

$$\left(\frac{\partial \boldsymbol{G}}{\partial \boldsymbol{X}} \right)^T \boldsymbol{\lambda} = \left(\frac{\partial l}{\partial \boldsymbol{X}} \right)^T. \tag{2.50}$$

Investigating the structure of Eq. 2.50, we find that the adjoint only requires solving of a *linear* system of equations, even if $\boldsymbol{G}$ (and therefore $\boldsymbol{l}$) are nonlinear.

Finally, we can solve for $\boldsymbol{\lambda}$ and substitute Eq. 2.50 in the loss sensitivity equation Eq. 2.49 by $\boldsymbol{\lambda}$, that now states [20]:

$$\frac{dl}{d\boldsymbol{p}} = \frac{\partial l}{\partial \boldsymbol{p}} - \boldsymbol{\lambda}^T \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{p}}. \tag{2.51}$$

So instead of solving the original loss function gradient in Eq. 2.20, we derived an alternative representation for the same gradient in Eq. 2.51. If we investigate Eq. 2.51, we can conclude that the adjoint can also be expressed as [20]

$$\boldsymbol{\lambda}^T = -\frac{\partial l}{\partial \boldsymbol{G}}. \tag{2.52}$$

Further, if we insert this into Eq. 2.51, we find a loss function gradient, that is similar to the original one in Eq. 2.20 but with the difference, that we take a different path — via $\boldsymbol{G}$ instead of $\boldsymbol{X}$ — for differentiation. Whereas the direct computation via $\boldsymbol{X}$ requires determination of the expensive Jacobian $\partial \boldsymbol{X} = ODESim(\boldsymbol{p}) / \partial \boldsymbol{p}$ (by solving a nonlinear system of equations), the computation via adjoint only requires determination of the adjoint by solving a linear system and the Jacobian $\partial \boldsymbol{G} / \partial \boldsymbol{p}$. To summarize, determination of the discrete adjoint only requires solving a linear system of equations (s. Eq. 2.50), and the dimension of this system only depends on the number of states and solver steps, and not on the number of parameters to determine sensitivities for.

Events

The discrete adjoint sensitivity analysis can be extended to also handle events, which is discussed for smooth events [29], [30], as well as for the generic case of events that cause *jumps* [31], [32]. The core method grounds on two major observations. First, analyzing discontinuous systems results in discontinuities in the adjoint trajectory as well [31], [33], [34], that need to be implemented. Second, as for the continuous adjoint, the event time now depends on the state and therefore on the parameterization, which requires the correction of the computed gradient [30], [31]. Since these procedures have already been implemented successfully (s. Sec. 2.3.2) and are not relevant for following this thesis, we will not go into further detail here.

2.1.3.6 Summary

It is worth mentioning that even if they look different at first glance, there is a strong relationship between some of these approaches. In Sapienza *et al.* [20] it is concluded that well-implemented reverse AD yields the same computations as the discrete adjoint method [35], and forward AD is consistent with the forward sensitivity method, at least for Runge-Kutta methods, however, this does not necessarily cause identical results. It is shown in multiple benchmark examples [19], that for problems with approximate sizes $|\boldsymbol{x}| + |\boldsymbol{p}| < 100$ forward-mode AD outperforms reverse-mode AD, as well as continuous methods. This cut-off value was later adopted to $|\boldsymbol{x}| + |\boldsymbol{p}| < 50$ because of improvements in methods and implementations [20]. For larger problems, the continuous adjoint approach scales better at the cost of a higher susceptibility to an unstable solving process.

Besides performance, another important point is the *stability* (in terms of solvability) of the approaches. Rackauckas *et al.* [36] investigate, that the discrete methods are found to be more stable, while continuous methods are faster but less stable. However, it is worth mentioning, that instability during solving of adjoint problems can be reduced or avoided by additional measures [37]. Stability issues where also examined during working on use cases for this thesis when applying the continuous adjoint method. Within this work, we therefore use a discrete reverse approach — the discrete adjoint method paired with AD — for sensitivity analysis of loss function gradients containing the solution of event ODEs. Nevertheless, it is important to have an overview of the available methods. Sapienza *et al.* [20] nicely conclude, that SA working with large systems still *"requires tailored approaches"*. This again emphasizes that the applicability of SA to models of industrial scale is still an active subject of research. The current state of research still makes it necessary to compare different approaches for the own application. The Julia SciML-ecosystem (s. Sec. 2.3.2) offers a very valuable implementation foundation here.

2.2 Terminology

Caused by the multidisciplinary nature of SciML, this thesis lives in (at least) two different worlds: ML and SciCo, which has a strong relation to engineering. The common bridge between both worlds is — as always — mathematics. However, terminology is not always clear, and the same identifiers are used stricter, more loosely, or entire differently in both domains. Because of this, some common terminology used in this work is introduced shortly in the following.

2.2.1 Models and parameters

A (simulation) model is defined as mathematical representation of a system or process that is often configurable, e.g. regarding its parameters [38]. In the following, we classify the models used within this thesis according to the attributes defined by Birta and Arbez [39] and complemented by Center [40]:

Static / Dynamic Mathematical models of a dynamic systems can predict one or more trajectories over time (dynamic) or only the equilibrium position (static) that is reached for $t \to \infty$. In this thesis, we focus especially on **dynamic** models.

Linear / Nonlinear If all equations within a model are linear, the model is designated a *linear* model and *nonlinear* if not. Deep machine learning models are nonlinear by design and because many physical effects (like e.g. air friction) are nonlinear, most physical simulation models are, too. This is the reason why in this thesis only **nonlinear** models are investigated. However, the proposed methods can be applied to linear models without restrictions, of course.

Deterministic / Stochastic Some models are deterministic, meaning an identical simulation result can be obtained by repeating the simulation. Stochastic models include stochastic effects, resulting in a different simulation result if the

simulation is performed repeatedly[10]. In this work, we focus on **deterministic** models. However, methods can be extended to stochastic models, which is future work (s. Sec. 9.3).

Continuous / Discrete Continuous models produce simulation results — so the state vector over time — that are continuously differentiable. Discrete models on the other hand, only change at specific points in time (the so-called *events*), which directly results in non-differentiable solution trajectories. In this work, we focus on **mixed continuous-discrete** models, so models that feature a continuous as well as a discrete part. This allows for modeling a width range of different physical effects.

Spatially-averaged / -distributed The real physical world is spatial distributed, which results in the fact that many computer simulations are modeled as such. For example in flood dynamics, it is quite common to use PDEs as simulation models to express spatial distributed physical effects (like e.g. wave convection and diffusion). However, in other physical domains like e.g. multi-body mechanics, it is common to replace computational expensive spatial-distributed by spatial-averaged models, that can be expressed by ODEs. Within this thesis, we work primarily with **spatially-averaged**, but **spatially-distributed** models as well.

Empirical / Mechanistic Mechanistic (or conceptual) models are derived from the observation of physical processes, so either by use of physical equations or by directly modeling a mechanistic behavior. If mechanistic models are derived directly from physical core principles, like e.g. conservation of energy, they are referred to as *first principles models*. Empirical models, on the other hand, are derived from an observation. Such models describe the behavior between physical quantities without representing the physical process behind that behavior in an explainable and generalizable way. Because mechanistic models are common in physics and empirical models are common in ML, we focus on both — **empirical and mechanistic** models — within this work.

Measured-parameter / Fitted-parameter Most simulation models are parameterizable, meaning their behavior can be changed by modifying their parameters. Physical models are often parameterized by doing experiments or measurements, resulting in a measured-parameter model. Most ML models on the other hand, do not provide interpretable parameters out-of-the-box and parameters are therefore *fitted*. However, parameter-fitting is also applied for physical models with less interpretable or hard-to-measure parameters. As a consequence, we handle models that include **measured** as well as **fitted parameters**.

In the following, we investigate example models from different domains and compare the terminology. Consider the following model, defined by an (algebraic) equation:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{u}(t), \boldsymbol{p}). \tag{2.53}$$

[10]If the stochastic process is modeled with random numbers, one could freeze the random seed to obtain a deterministic simulation result, of course.

Note, that parameters $\boldsymbol{p}$ are constant over time whereas the inputs $\boldsymbol{u}(t)$ are not. This results in a model with time-dependent output $\boldsymbol{y}(t)$. This function $\boldsymbol{g}$ may be based on a system of equations, that is derived from physical principles, in this case we call it *physical model* and its parameters *physical parameters*. An example for a physical model is the kinetic energy formalism in classical mechanics:

$$e_{kin}(v(t), m) = \frac{1}{2} m v(t)^2, \tag{2.54}$$

with input velocity $\boldsymbol{u} = [v]$, parameterizable mass $\boldsymbol{p} = [m]$, and output the kinetic energy $\boldsymbol{y} = [e_{kin}]$. Note, that for such *physical* systems, we have a very good understanding of the parameters and the variables in general.

If the equation $\boldsymbol{g}$ has no physical foundation and aims at approximating unknown functions, it is called a ML model and the parameters are referred to as *machine learning parameters* respectively. An example for a ML model is the fully-connected feed forward layer:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{u}(t), \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{\sigma}(\boldsymbol{W}\boldsymbol{u}(t) + \boldsymbol{b}). \tag{2.55}$$

Here, $\boldsymbol{\sigma}$ is an activation function, $\boldsymbol{u}$ the layer input, $\boldsymbol{W}$ the weight and $\boldsymbol{b}$ the bias parameters.

In the following, we especially focus on physical models as representative of the class of mechanistic/conceptual models and on machine learning models as representative of empirical models. We summarize that, from a mathematical point of view, there is no obvious difference between physical and ML models. Both types refer to a *parameterizable* system of equations which calculate an output $\boldsymbol{y}$ based on parameters $\boldsymbol{p}$. Besides the origin of the equation system — physical equations vs. universal approximators (UAs) — the major difference is how the corresponding parameters are interpreted and determined. For physical models, $\boldsymbol{p}$ has a physical meaning and units and is determined by investigating the real physical system. For ML models, $\boldsymbol{p}$ has (in most cases) no physical meaning and is determined by mathematical optimization. While the observation that both model types can be treated equally may seem trivial at first glance, there are countless publications in the field of machine learning that rediscover basic mathematical methods and only apply them to ML instead of physical models.

Time

In addition to inputs, some models *explicitly* depend on time. Whereas this is quite uncommon for pure physical systems, sometimes it is easier to model a time dependency than the correct physical input. Consider the following example: The output for a solar panel shall be predicted for a given time of day. Physically correct, the model should take the current angle between the sun and the solar panel as input. However, a model that just takes the time as input may be far easier to set up (and use later) — even if physically less accurate. The explicitly time-dependent model can be obtained by adding a function argument t:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{u}(t), \boldsymbol{p}, t). \tag{2.56}$$

Variability

Variables can be classified more precisely according to the situation at which they

change. For variables in our models, the following *variabilities* can be distinguished [41]:

Continuous The variable value changes over time and is always *continuously differentiable*. The derivative is always finite.

Discontinuous The variable value changes over time, but is not *continuously differentiable*. This is because there are jumps with infinite derivatives between continuous parts.

Discrete The variable value changes only and immediately at an event instance. The time derivative is always zero, except for the event instances where it is zero (no jump) or infinity (finite jump).

Constant The quantity doesn't change at all w.r.t. the independent variable (e.g. time). The derivative is therefore always zero.

Similarly, functions can be classified regarding the variable they are calculating and equations regarding the variable they are solved for. Fig. 2.4 shows the different variabilities next to each other.

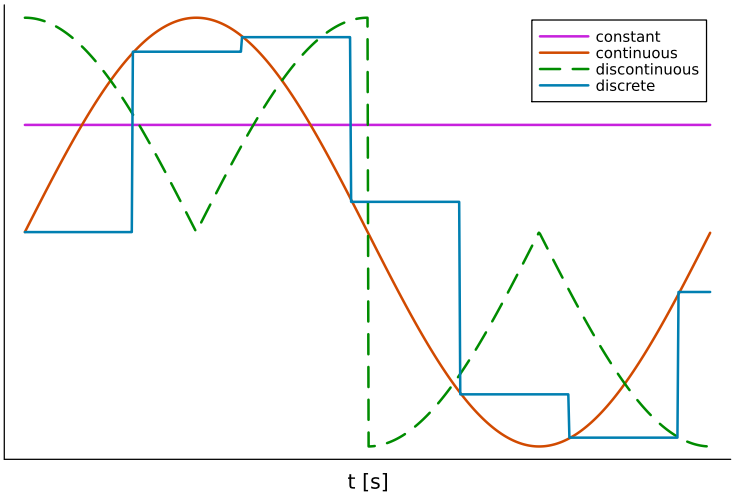


Figure 2.4: Demo curves showing the different types of variability.

Deep models and deep learning
In ML, a model is considered *deep*, if multiple layers of ANNs are stacked together. In this thesis, almost every ML model is *deep*, therefore this prefix is skipped and non-deep architectures are explicitly identified as *single-layered* or *shallow*.

2.2.2 States

Some models — and these are especially relevant within this thesis — not only rely on inputs, parameters, and time, but also on a *state* $\boldsymbol{x}$:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{u}(t), \boldsymbol{p}, t). \tag{2.57}$$

In general, the system state is a set of values that uniquely represents the situation of the system, meaning the future advancement of the system is known for a given state. Whereas the choice of states from a pool of variables is not necessarily unique, a valid choice further does not include any redundant entries, which would lead to a singular system. Depending on the domain, updating the states takes place differently. An example of a state dependent physical model is the *translational pendulum*, given by the ODE

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{p}, t) = \begin{bmatrix} v(t) \\ a(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ \frac{-c \cdot s(t) - d \cdot v(t)}{m} \end{bmatrix}, \tag{2.58}$$

with state $\boldsymbol{x} = [s\ v]^T$ (position and velocity), state derivative $\dot{\boldsymbol{x}} = [v\ a]^T$ (velocity and acceleration) and parameters $\boldsymbol{p} = [c\ d\ m]^T$ (spring constant, damper constant and mass). Within physical simulations, states are solved by numerical integration of the state derivative (s. Sec. 2.1.2).

An example for state-full models in ML are recurrent neural networks (RNNs). The RNN layer output is given by the equation:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{x}_d(t), \boldsymbol{u}(t), \boldsymbol{p}) = \boldsymbol{\sigma}(\boldsymbol{W}_u \boldsymbol{u} + \boldsymbol{W}_x \boldsymbol{x}_d + \boldsymbol{b}), \tag{2.59}$$

with parameters $\boldsymbol{W}_u, \boldsymbol{W}_x, \boldsymbol{b} \in \boldsymbol{p}$ and $\boldsymbol{\sigma}$ the nonlinear activation function. In ML, usually update rules are defined, which means the state is updated at specific points in time. For RNNs, the state $\boldsymbol{x}_d$ is updated according to the following rule:

$$\boldsymbol{x}_d(t_{i+1}) = \boldsymbol{y}(t_i), \tag{2.60}$$

meaning the state at the next time instant equals the output of the current time instant. In this case, the state acts as the *memory* of the previous output. Some ML models (such as the RNNs above) use the term *hidden state*. It is important to note that in general, the hidden state does not match the *continuous state* definition used in this thesis, which is driven by the systems engineering point of view. Therefore, a different symbol $\boldsymbol{x}_d$ is used for the hidden state, while the actual state of the system is denoted by $\boldsymbol{x}$. In fact, the hidden state only changes at discrete points in time, and is therefore referred to as *discrete state*. In Sec. 3.3, this relationship is further investigated and a connection between the two state definitions is derived.

2.3 Programming language, software and standards

Within this section, we want to introduce topics beyond pure mathematics, that are relevant for understanding this work. This captures the software standard FMI (see Sec. 2.3.1) and the Julia programming language (see Sec. 2.3.2).

2.3.1 Functional mock-up interface (FMI)

The functional mock-up interface (FMI) is a container format for simulation models of industrial scale [41], [42], [43]. At the time of writing, FMI version 2 is still the most used in everyday engineering. However, in 2023 the FMI version 3 was released and will successively be adapted in the practical application. The version 1

is outdated and almost not used anymore. Models exported under the requirements of the standard are referred to as functional mock-up units (FMUs). From a technical perspective, functional mock-up units (FMUs) are ZIP archives, including a standardized model description file (as XML) together with binaries and/or C code of the simulation model. Binaries for multiple platforms can be included to support multiple computer system architectures. The FMI defines three types of interfaces for models, that can be supported by a FMU. Any FMU can support either only one or more of these interfaces, that we further investigate in the following.

Model exchange (ME)
The FMU interface model exchange (ME) is the most relevant for the use cases considered in this thesis. ME-FMUs provide the interface of an event ODE (s. Fig. 2.5a), meaning an interface in the form of $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{p}, \boldsymbol{u}, t)$, including further functionality for handling of events. As a result, this type of FMU needs to be embedded into a simulation framework featuring an appropriate numerical solver with support for event handling to obtain a solution. While the implementation of such a simulation framework requires some effort, the low-level ME interface allows for manipulation of the model dynamics before numerical integration by extending this simulation framework, which enables interesting modification scenarios as further investigated in Sec. 7.3. Whereas FMUs with other interface types, like the interface co-simulation (CS) introduced in the following, could also be extended to allow modification of the state derivative, this would require additional arrangements during modeling.

Co-simulation (CS)
Compared to ME-FMUs, the co-simulation (CS) interface is easier to use from the perspective of developing the simulation environment. The model interface is in the form $\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{p}, \boldsymbol{u}, t)$ (see Fig. 2.5b), so no numerical integrator is needed, and a solution can be obtained directly. In fact, the numerical integrator, event handling, as well as the system state is part of the FMU itself[11] and hidden from the user.

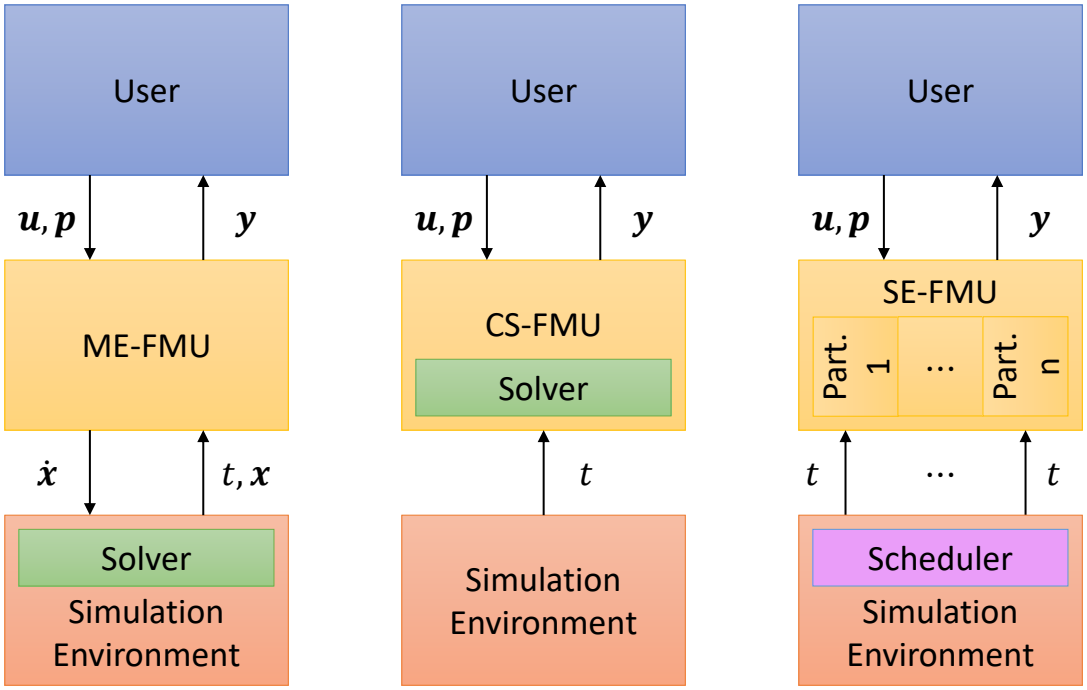
The native integration of a solver results in two things. First, the FMU can contain not only algebraic or discrete systems of equations, as well as ODEs, but also PDEs or even DAEs, together with a suitable solver for the problem type. Second, while this allows for a much easier simulation environment, it comes with the price of much more restricted application scenarios. For example, we find that we cannot easily modify the system state derivative, because it is not available from the CS interface definition. Neither the states of the system, nor the state derivatives are accessible out-of-the-box. These signals could be added manually as inputs and outputs when creating the model, but this requires deep knowledge[12] of the model and additional implementation effort.

Scheduled execution (SE)
Scheduled execution (SE) is a novel simulation mode that was added in FMI version 3. In general, all FMI simulation modes allow the model to consist of multiple

[11] In case of an ODE system. Strictly speaking, the FMI definition does not force that the solver is part of the FMU, but in practice this is almost always (and always within this work) the case.

[12] For example in object-oriented modeling (or DAEs in general) one needs to know beforehand (or manually control it) which variables will be states after model compilation and index reduction.

partitions. The selection of the model partition is called *scheduling*, and the corresponding algorithm or strategy is called *scheduler*. In ME and CS, scheduling of the model partition must occur within the FMU. For models that are separated into multiple *model partitions*, SE allows the *simulation environment* to control which partition is executed during simulation. Because SE is a relatively new feature, it is not implemented in many tools for now and therefore not further investigated in this thesis. However, it seems a promising direction of research for future work.



(a) For **model exchange (ME)**, the solver is part of the simulation environment. The scheduler (optionally, not drawn) is part of the FMU.

(b) For **co-simulation (CS)**, the solver is (optionally, but usually) part of the FMU, as well as (optionally, not drawn) the scheduler.

(c) For **scheduled execution (SE)**, the solver is part of the FMU (optionally, not drawn), that consists of n model partitions. The scheduler is part of the simulation environment.

Figure 2.5: The different simulation interfaces defined in FMI.

2.3.2 Julia programming language

The Julia programming language (in the following abbreviated with *Julia*) is a relative young (2012) platform-independent programming language [44], [45]. However in 2023, Julia was able to hit the *TIOBE Index Top 20*, listing the top 20 programming languages [46]. The programming style is *high-level*, which is strengthened by the availability of many problem-oriented programming libraries for typical implementations, like for example for linear algebra (*LinearAlgebra.jl*) or the numerical solving of DEs (*DifferentialEquations.jl* [14]). Memory allocation is performed automatically, the same applies for memory freeing, which is performed by integrated garbage collection. The syntax of the language is close to other state-of-the-art high-level programming languages like Python or Matlab.

Even if Julia is not an object-oriented language, one of its key features, the *multiple dispatch*, allows for a similar code organization then in object-oriented programming. In multiple-dispatch, multiple functions can be defined sharing the same name and even the same number of parameters. Whenever the function identifier is called, the function with the best fit, based on the *types* of the function arguments, is selected and executed. However, this introduces the problem of *function ambiguity*, meaning there might be multiple function candidates that fit *equally well*. Further, Julia even allows for the definition of functions independent of the argument types, which is refereed to as *parametric polymorphism*.

Besides its flexibility and usability, Julia is able to produce very fast executing code compared to other high-level languages like Python. This is accomplished by just-in-time (JIT) compiling of Julia code to machine code. Typically for this feature, this introduces compilation time for the first execution of code, however there exists multiple ways[13] to reduce or oven prevent this time delay.

Finally, Julia provides interfaces to other programming languages like Python and a seamless integration for calling C-functions via `ccall`. This is especially important for the integration of FMI (s. Sec. 7.3).

The SciML ecosystem

The SciML ecosystem [36] within the Julia programming language is a collection of software libraries to allow for the creation and training of universal differential equations (UDEs), that are further introduced and discussed in Sec. 3.3.1. It features a full integration of libraries for ML (*Lux.jl* [47], [48] and *Flux.jl* [49], [50]), solving of differential equations (*DifferentialEquations.jl* [14]), automatic differentiation (*Enzyme.jl* [51], [52], *Zygote.jl* [53], *ForwardDiff.jl* [54] and *ReverseDiff.jl*) and many more. Because the Julia SciML framework is designed based on very generic requirements (UDEs), it allows for the development and implementation of a variety of existing — and even not yet discovered — architectures for a width range of use cases beyond specific application domains. The ecosystem is still in active development and is extended on a daily basis by the developer community.

2.4 Recurring example: Bouncing ball (BB)

The methods in the thesis are validated at real industrial applications from different projects. However, for the deduction of methods and illustration of the functionality, a more simple but yet not trivial model is used: A 2D model of the well-known bouncing ball, including nonlinear friction. In the following, this model is briefly introduced.

The bouncing ball consists of four continuous states, the position in both spatial axis $\boldsymbol{s} = [s_x \; s_y]^T$, and the corresponding velocities $\boldsymbol{v} = [v_x \; v_y]^T$, all together:

$$\boldsymbol{x}_c = [s_x \; v_x \; s_y \; v_y]^T. \tag{2.61}$$

Because we focus on systems containing discrete subsystems, the continuous dynamics are extended by a discrete state n, that is incremented whenever the ball hits

[13] For example, static compilation with `julia-c` (Julia version 1.12) or dynamic compilation with *PackageCompiler.jl*.

one of the walls (*collision counter*):

$$\boldsymbol{x}_d = [n]. \tag{2.62}$$

This discrete state influences the dynamics of the bouncing ball. As soon as a pre-defined threshold of collisions is reached, the ball *breaks*, which is modeled by significantly increased air-resistance — like a popped balloon. The model is *parameterizable*, featuring the following parameters and default values:

- The mass of the ball $m = 1\,\mathrm{kg}$,
- the gravity constant $g = 9.81\,\mathrm{m/s^2}$,
- the radius of the ball $r = 0.1\,\mathrm{m}$,
- the percentage energy loss by collision $d = 10\,\% = 0.1$,
- the number of collisions before the ball breaks $n_{max} = 10$,
- the air resistance coefficient $\mu_{ok} = 0.15$ for the healthy ball with $n < n_{max}$ and
- the air resistance coefficient $\mu_{brk} = 1.5$ for the "broken" ball where $n \geq n_{max}$.

The right-hand side of the ODE for the *healthy* bouncing ball is then given by:

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{p}) = \dot{\boldsymbol{x}} = \begin{bmatrix} \dot{s}_x \\ \dot{v}_x \\ \dot{s}_y \\ \dot{v}_y \\ \dot{n} \end{bmatrix} = \begin{bmatrix} v_x \\ \frac{-|\boldsymbol{v}| \cdot v_x \cdot \mu_{ok}}{m} \\ v_y \\ \frac{-|\boldsymbol{v}| \cdot v_y \cdot \mu_{ok}}{m} - g \\ 0 \end{bmatrix}, \tag{2.63}$$

whereas μ_{ok} is replaced by μ_{brk} for the case of the broken ball with $n \geq n_{max}$. The ball moves inside a square room of size 2 m centered at the coordinate frame origin and is able to collide with the four walls. Whenever a collision happens, the perpendicular component of the velocity vector with respect to the surface is negated, resulting in four cases for collisions that need to be distinguished:

Bottom for $1\,\mathrm{m} + s_y - r = 0$, velocity after impact $-v_y \cdot (1 - d)$.

Top for $1\,\mathrm{m} - s_y - r = 0$, velocity after impact $-v_y \cdot (1 - d)$.

Left for $1\,\mathrm{m} + s_x - r = 0$, velocity after impact $-v_x \cdot (1 - d)$.

Right for $1\,\mathrm{m} - s_x - r = 0$, velocity after impact $-v_x \cdot (1 - d)$.

An exemplified solution of the bouncing ball is given in Fig. 2.6.

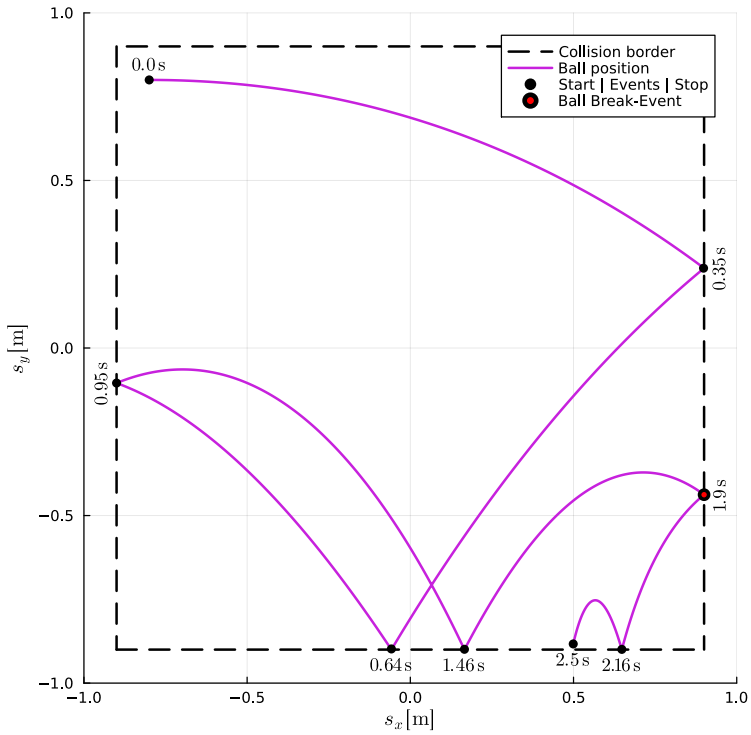


Figure 2.6: The position of the bouncing ball (violet) in two dimensions from side view for the introduced default parameterization. The start state is $\boldsymbol{x}_c(t_0) = [-0.8\,\mathrm{m} \quad 5.5\,\mathrm{m/s} \quad 0.8\,\mathrm{m} \quad 0.0\,\mathrm{m/s}]^T$ and $\boldsymbol{x}_d(t_0) = [5]$, and it is simulated for 2.5 s. Gravity is accelerating the ball in y-direction. The ball is slowed down by air friction. Whenever it collides with one of the four walls, a fraction of energy is lost. Such events are marked (black dot) and the corresponding event time is annotated. Starting from the event where the ball breaks (red dot), the air friction drastically increases.

The model features multiple advantages:

- The core concept of the bouncing ball is very common for many researchers and engineers in the field, there is a general intuition of simulation results and the system behavior.
- Simulation results can be visualized very well understandable by plotting the ball position in directions x and y, which corresponds to a side-view on the actual physical system. This allows an easy and fast interpretation of results.
- If the air friction coefficient is chosen to be zero, the right-hand side of the system is fully linear. For non-zero friction coefficients, it is nonlinear. This allows for easily scaling the model complexity.
- The system offers *real* discontinuities on state level by changing the ball velocity. Note, that in general systems with events don't necessarily need to change their continuous state at event instances.
- By introducing n_{max}, the system becomes dependent on the discrete state n, so that the ball dynamics suddenly change if the collision threshold is met. Such configurations are often referred to as *modes*, so in this case the *OK*

and *broken* modes of the system. However, for large values of $n_{max} >> 0$, the breaking condition is not met in the practical simulation and the system dynamics are independent of the number of happened collisions and therefore the discrete state.

Equipped with this example model, we can start into the actual methodological chapters.

Chapter 3

Creation

Figure 3.1: The modern simulation engineer is struggling with the decision: *Shall I model physics-based or with machine learning?* He comes to an interesting decision... (This image was generated with the assistance of AI).

A variety of synonyms exist for the process of combining physical models, so models that are based on the observation of physical effects, and machine learning models: *Hybrid Modeling* [9], [55], [56], *Greybox-Modeling* (the British spelling) [57], *Graybox-Modeling* (the American spelling) or *Semi-Physical Modeling* [58], *Semi-Mechanisitic Modeling* [59], *Neuro-Mechanisitic Modeling* [60] and *Integrated Neural Network Modeling* [61] — to name only a few. Nevertheless, none of the terms has yet been able to clearly establish itself. One of the reasons for this is that often new terms are introduced, depending on which components are combined exactly: While *mechanistic*, *white-box* or *first-principles models* are often used on the side of physical modeling for example, the degree of abstraction is varied on the machine learning side, which is reflected in the use of very specific (such as FFNNs [61]) to very generic model classes (such as UAs [36]), or even more abstract, *neural networks* or *data-driven models*. Unfortunately, this variance in designators for the involved models creates an ideal basis for getting creative when naming the combined research field, and makes capturing the development and state of the art challenging. Because the term is one of the more frequently used ones and particularly popular in the industrial environment, we proceed to use the term *hybrid modeling* in this thesis.

3.1 Introduction

Probably one of the most frequently used expressions in research is the term *hybrid*, usually in the context that two (or more) things merge and something better evolves from this. Even in the limited scope of this work, there are at least two different meanings of the term *hybrid model*. In the field of DEs, a hybrid model (HM) refers to a dynamic system that features continuous, as well as discrete dy-

	Physical model	ML model
Correctness (physically)	yes, under model assumptions	only an approximation
Accuracy	limited by modeling simplifications (*sim-to-real-gap*)	unlimited
Extrapolation	good, as long as modeling assumptions are valid	bad (in general)
Interpretability	excellent, values have a human understandable meaning, physical units and intuitive sensitivities	modest, values are pure numeric and have no physical meaning (except values that are trained to match physical quantities)
Modeled by	simulation engineer	ML scientists

Table 3.1: A high-level comparison of physical and ML models.

namics. In other domains, and within this thesis, the term *hybrid modeling* refers to the combination of one or more physical and ML models. The goal of this symbiosis is to unite advantages of both modeling approaches. In the following, we compare the attributes of both, that are interestingly almost contrary (see Tab. 3.1), and ignore very exotic model types that may lead to outliers.

While physical models are by design *correct* — the equations are derived from first principles that were proven over centuries as part of countless scientific experiments — and are able to extrapolate within the validity of the derived equations[1], MLs models only approximate these equations and show serious difficulties if operated beyond the training data space. However, high dimensional ML models are able to approximate arbitrary functions with any accuracy needed, which makes them an ideal candidate to learn for hard to model and/or hard to solve physical equations. A direct result of the physical nature of physical models is that they are much easier to interpret: Not only the equations and their parameters can be understood, but also the variables connecting submodels have physical units and a human understandable meaning. This further results in a good intuition regarding sensitivities[2]. For ML models it is the other way round, neither values nor sensitivities can be interpreted in a physical context in general. The only exception (but with limitations) are model outputs, that are trained to match physical measurements. Depending on the modeling type, a very different skill set is required, which is why very different experts are involved in their creation: Physical models are designed by *simulation engineers*, whereas MLs models are built by *ML scientists* in general. This already shows how differently far (engineer vs. scientist) these two scientific

[1]Technically, physical equations are nothing but a fit of a mathematical model (at a chosen level of abstraction) to a huge dataset: The records of countless experiments of the past, that basically lead to the deduction of these equations.

[2]Model engineers have a good intuition what happens, if they increase or decrease values of their simulation model. During development, they do many finite differences approximations of sensitivities by hand (by changing values and examining the consequences).

fields are integrated into industrial practice. An unpleasant side effect of the two types of experts involved, is an inconsistent terminology, resulting in the same terms being used in different contexts, as discussed in Sec. 2.2. Interestingly, the naming conventions are more driven by the ML side of the research community than the physical modeling side: The starting point is the ML model, that is (structurally) *enhanced* [62] or its training process is *informed* [12], [63] by a physical model. This can be misleading, because often — and in this thesis' examples — the physical model part is proportionately larger[3] than the ML part. Only a few publications, such as Thuerey *et al.* [64], speak of *physics-based* deep learning, which more brings the physical part to the fore. Last but not least, both model types differ significantly in terms of the amount of data and prior knowledge required for creation, see Fig. 3.2.

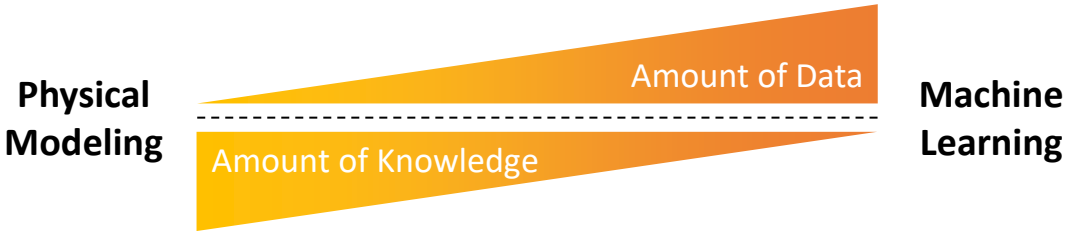


Figure 3.2: Physical modeling does not require data, but a detailed understanding of the system to be modeled. For a ML model it is exactly the other way around. Nevertheless, modeling purely on the basis of ML is only possible if a sufficiently large amount of data is available. If not, we need a backup plan ...

To conclude, the aim of combining physical models and ML models is to retain the positive characteristics of both approaches, without inheriting the negative characteristics, or only to a very limited extent. The desired attributes for the created hybrid model are the physical correctness, extrapolation capability and interpretability of physical models and the unlimited accuracy of machine learning models.

Besides the combination of physical models and ML models, (probably) the most fundamental approach to developing new methods in ML is the integration of multiple ML structures into each other, also here with the goal to unite (at least some) advantages. An example for this are ODE-RNNs, which combine a (neural) ODE, a RNN and an additional FFNN for the outputs. The goal behind this specific concept is to make RNNs more robust against irregular sampled input data and is discussed by Rubanova *et al.* [65]. Or Cao *et al.* [66] investigate the automatic combination of long short-term memorys (LSTMs) [67], gated recurrent units (GRUs) [68], Transformers [69], and state-space models, where a user defined preference function is added to a multi-objective optimization problem to determine an optimal number and sequence for the combined ML models. Even if the focus of hybrid modeling, and this thesis, lies on the combination of physical models and ML models, the combination of ML models with each other can be discussed side-by-side with little

[3]Comparing the size of ML models and physical models does not appear obvious. However, one could compare the number of modeled equations in the physical models to the number of *approximated* equations in ML models. In many applications, the number of approximated equations is not known, but can roughly be estimated in some cases.

additional effort, as is shown later.

From a methodical point of view, the main question associated with the field of hybrid modeling is for sure:

How can different models be combined?

Whereas this question could be answered by deriving a paradigm for hybrid modeling consisting of design rules, it is obvious that this paradigm heavily depends on the *type* of model, that should be allowed for hybrid modeling. So a further question is raised:

What kinds of models can (or need to) be combined?

Obviously, both questions are strongly connected, and a variety of valid hybrid modeling paradigms can be found, depending on which model types are investigated exactly. Of course the requirement to combine *physical* and *ML* models is not sufficiently accurate for a mathematical discussion. So in order to derive hybrid modeling design rules that are compatible with models from real life engineering, one must first answer the question:

What are the properties (or requirements)
of models used in real life engineering?

This question is answered in the following. After that, equipped with requirements, the state of the art of hybrid modeling can be discussed and the need for a new design paradigm can be derived.

Requirements of physical models in real life engineering
Capturing all requirements of simulation models, that are relevant to industrial practice, is a huge effort and not practicable. It is target-oriented to focus on a subset of models, that captures a *significant amount* of relevant use cases. Even the specification of this subset is a work on its own, therefore the following is (justified) assumed:

Premise 1 *A mathematical class of models, that is capable of describing the class of models that can be expressed by the FMI (so FMUs), is able to capture a* significant subset *of simulation models in real life engineering applications. More detailed, this describes the modeling of arbitrary hybrid (in terms of* discontinuous*) ODEs, as well as algebraic and discrete systems. This class is formally introduced later and referred to as mixed hybrid universal discrete algebraic ordinary differential equation (HUDA-ODE).*

This premise is based on the following two observations:

1. One of the main goals of FMI is to provide an interface for the exchange of industrial models that is applicable to a broad range of use cases. Therefore, the same underlying goal of *industrial relevance* is shared. Many of the best engineers from the most successful companies in the world worked for over two decades on finding a good solution. Chances are high, that they found one.

2. Without examining the industrial usage of FMI in detail and determining its relevance, it is not possible to quantitatively prove the premise (*FMI is relevant*) above. Statistically, the chances are good that it is true, because FMI is (at least one of) the most used model exchange format in industry with over 250 tools implementing it [42]. Further, all presented engineering use cases in this work (s. Cha. 8) satisfy the requirement being compatible with FMI, all corresponding models are implemented as FMUs. However, we need to clarify, that FMI targets on system simulation problems and even if it is very versatile[4], there might be more suitable solutions for specific application niches.

Requirements of ML models in real life engineering
Collecting requirements of machine learning models in engineering is more challenging, because ML is still not a widely used — or even accepted — technique in industrial development processes. ML is still deployed only for very specific modeling tasks. On the other hand, many machine learning models can be expressed as algebraic (e.g. FFNN), discrete (e.g. RNN) or continuously differentiable (e.g. neural ordinary differential equation (NODE)) function and therefore as FMU. This is investigated in more detail in Sec. A.2. In the following it is assumed that this set of ML models is sufficient to make this work industry-relevant.

In summary, we conclude that algebraic, discrete, and ordinary differential equation systems can be used to model a significant number of industry-relevant application scenarios, and that the combination of physical and ML models in an industry-relevant context can be discussed on this mathematical basis.

In the following Sec. 3.2, the state of the art for the combination of physical models and ML models to obtain hybrid models is discussed in the context of the gathered requirements from real life simulation models. The requirements behind these designs are investigated and the relevance of a new view on hybrid modeling is emphasized.

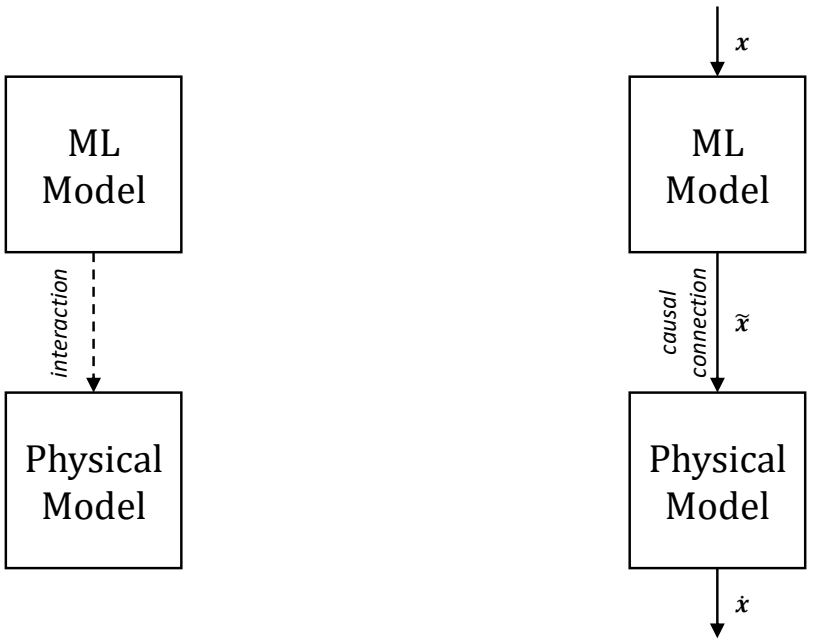
3.2 State of the art

The landscape of publications in the field of hybrid modeling can be subdivided in publications that present successfully processed use cases and thus offer an excellent starting point for own applications, and publications that consider the actual topic from a greater distance, detached from a specific application case. Publications like for example by Zendehboudi *et al.* [70] provide a great overview of the potentials of hybrid modeling and successfully processed use cases from the modeling of chemical and energy systems. This strengthens the relevance of the hybrid modeling approach and shows the broad applicability beyond a specific application domain. A very positive development is, that recently the number of publications concerning hybrid modeling in real world scenarios increases. Concerning specific applications, it is shown that applying hybrid modeling to packed bed thermal energy storage systems [71] and heat pumps [72] yields an immense speed-up while keeping high accuracy.

[4]Especially CS-FMUs can contain any type of DE, including 3D-PDEs and corresponding solvers.

Or in battery modeling [73], the accuracy can be significantly increased. Further, application scenarios from classic engineering are being opened up for hybrid models, e.g. hybrid models show striking benefits (speed-up and accuracy) if applied in model predictive control applications [74]. This is just a glimpse into a field of hybrid modeling applications that is currently experiencing strong growth.

We proceed by doing one step back and switching to a more methodological perspective. Across various publications, the combination of physical and ML models (*hybrid modeling*) is discussed, but on different layers of abstraction (s. Fig. 3.3). Most publications focus on the combination of model *blocks* that are causally connected with each other, as further investigated in Sec. 3.2.2. Further and in the following section Sec. 3.2.1, publications are presented that discuss the combination in a very general way at *application* level. We will show, that both perspectives are not sufficient to discuss hybrid modeling under industrial requirements and that a third perspective, the level of *equations*, is needed.



(a) **Application level**
Physical and ML model *interact* along the application. For example, the ML model is used to pre-process data for the physical model.

(b) **Model level**
Physical and ML model are *connected* in a *causal* way. For example, the ML model pre-processes the value $\boldsymbol{x}$ and connects the resulting quantity $\tilde{\boldsymbol{x}}$ to the physical model. This is only possible for causalized and continuous simulation models, as explained later in the chapter.

Figure 3.3: A view on the same hybrid model, consisting of a ML model and a physical model, from two different levels of abstraction. All presented publications (state of the art) stick to one of the two perspectives *Application* or *Model*.

3.2.1 Application level

In Rueden *et al.* [75], the combination of models from physical modeling and machine learning is discussed in a very generic way, comparing where (the location)

and when (in the process) simulation and ML can be combined. It is distinguished between *simulation-assisted ML* (simulation as part of the training data, hypothesis, algorithm or verification) and *ML-assisted simulation* (ML models as part of simulation models, to determine model inputs/parameters, as part of numerical methods or processor for simulation results). The term *hybrid model* is suggested to be used only if this distinction cannot be made because of the advanced, deep symbiosis of both concepts. The paper investigates *possibilities* to integrate simulation and ML (and the other way round), but does not investigate specific structures, making this contribution a good overview over general possibilities on *application level*.

Discussion
Discussing hybrid modeling on the level of application provides a valuable starting point. Very general, but far-reaching decisions, like the roles of physical and ML models in the application, can be made on this level. In the actual application, however, no concrete statements can be made about the technical implementation, for example which signals are (or can) be connected and what consequences result from this.

Because this work explicitly focuses on the technical feasibility of hybrid modeling, hybrid models are investigated on the level of combining model "blocks" in the following.

3.2.2 Model level

Starting over 30 years ago, the term *hybrid model* was used for the first time by Psichogios and Ungar [9] for chemical process modeling [7] on *model level*. However, the original idea of incorporating a priori knowledge into ANNs was published at least one year earlier [76]. Since then, many further publications, especially from the field of modeling chemical reactors, adapted the term. In the following, different hybrid modeling architectures on *model level* are introduced. This concerns the combination of *model blocks*, that express physical models as well as ML models and connections between them.

Serial and parallel structures
Even if proposed around three decades ago [77], the serial and parallel structure of hybrid modeling are still part of the state-of-the-art methods for combining models, as very recent publications like Rudolph *et al.* [78] show. This is because these patterns are defined in a very generic way and are therefore applicable to a wide range of use cases. Because of their simplicity, they offer a versatile starting point to derive more complex architectures. Multiple publications [70], [77], [78], [79] propose a generic serial or sequential (s. Fig. 3.4 and 3.5) and parallel (s. Fig. 3.6) combination of models. In the context of specific application domains and use cases, this is also discussed in numerous further publications like in Jirasek and Hasse [80] for thermodynamic modeling of fluid mixtures or in Leifsson *et al.* [81] at the example of modeling ocean vessels.

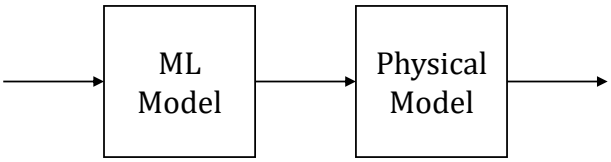


Figure 3.4: The serial architecture (ML model first) for hybrid modeling, for example proposed for ANNs by Thompson and Kramer [77] (adapted figure). The ML model manipulates *values*, before passing them to a physical simulation model. This is also referred to as *feature learning model* [78].

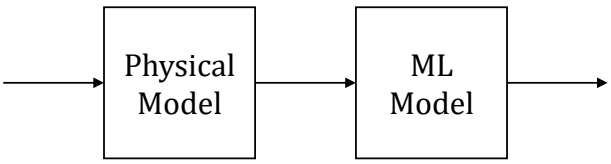


Figure 3.5: The serial architecture (physical model is inferred first) for hybrid modeling, for example proposed by Rudolph *et al.* [78] (adapted figure). The physical model manipulates *values*, before passing them to a ML model. This is also referred to as *physics-based preprocessing model* [78].

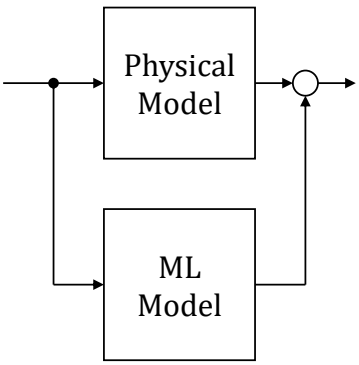


Figure 3.6: The parallel architecture for hybrid modeling, for example proposed for ANNs by Thompson and Kramer [77] (adapted figure). The ML model learns for *residuals* and adds them to the model output. This is also referred to as *delta model* [78].

Advanced / Specific structures
With the aim of getting a good overview of relevant architectures in addition to the pure serial and parallel patterns, more recent as well as historical (but still relevant) design patterns are examined in the following. Multiple architectures are compared by Duarte *et al.* [57]: Parameter estimation via ANN [9], [56] (s. Fig. 3.8), Residual learning with a RNN [61] (s. Fig. 3.9), and multi-part hybrid models [77] (s. Fig. 3.10). Further, Lai *et al.* [82] propose a state-of-the-art hybrid modeling concept that changes derivatives before numerical integration (s. Fig. 3.7).

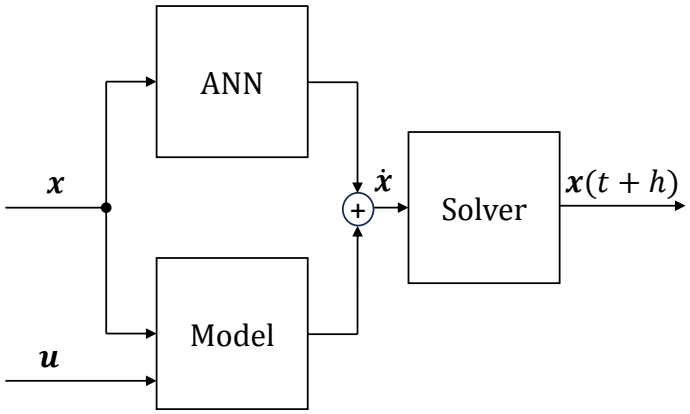


Figure 3.7: *Physics-informed neural ODEs* [82]: The ANN and the (physical) model both compute a state derivative based on the same system state $\boldsymbol{x}$. The individual state derivatives are added and the resulting system state derivative $\dot{\boldsymbol{x}}$ is integrated by a numerical solver. It is important to note that within this work we use the term *informed* differently — not for structural integration but for regularization.

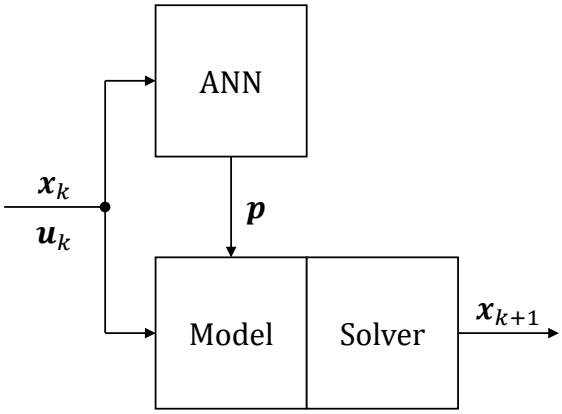


Figure 3.8: *ANN as parameter estimator*: Hybrid model architecture proposed by Psichogios and Ungar [9] (adapted figure). Based on the current state $\boldsymbol{x}_k$ and input $\boldsymbol{u}_k$, the ANN computes parameters $\boldsymbol{p}$ that are passed on to the simulation model, which computes the next system state $\boldsymbol{x}_{k+1}$. A similar architecture is proposed by Schubert *et al.* [56], but extended by a *Fuzzy Expert System* that is able to influence the system state before the next simulation step, see Fig. A.1.

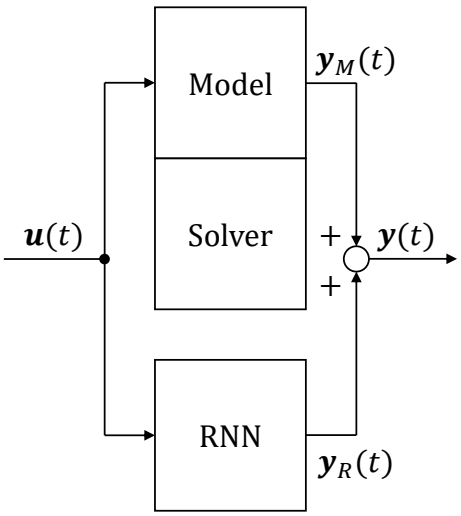


Figure 3.9: *Residual learning with RNN*: Hybrid model architecture proposed by Su *et al.* [61] (adapted figure). Based on a system input $\boldsymbol{u}$, the physical model gives a system output $\boldsymbol{y}_M$. The residual (the model deviation) $\boldsymbol{y}_R$ is predicted by a RNN, which was trained on this task, and added to the model output $\boldsymbol{y}$.

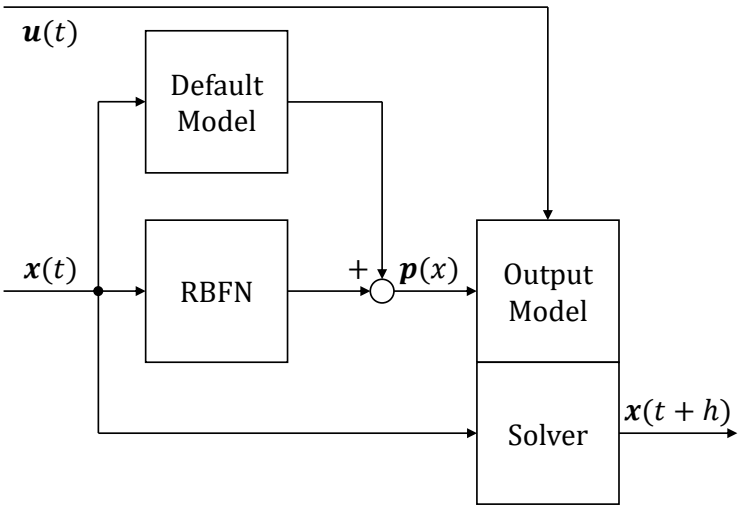


Figure 3.10: *Multi-part hybrid model*: Hybrid model architecture proposed by Thompson and Kramer [77] (adapted figure). The *default model* calculates parameters $\boldsymbol{p}(\boldsymbol{x})$ based on the system state $\boldsymbol{x}$ together with a radial basis function network (RBFN). The *output model* holds the actual system dynamics, that further depend on an input $\boldsymbol{u}$, and calculates the next system state $\boldsymbol{x}(t+h)$ paired with a numerical solver.

Discussion
Based on the presented parallel and sequential, as well as more advanced architectures, we recall the original question behind hybrid modeling:

How can different models be combined?

We find, that hybrid modeling on *model level* is indeed capable of answering this question by providing a variety of possible combinations and combination schemes. Some general aspects of state-of-the-art as well as historical hybrid modeling can be observed:

- Different *layouts* are applied, that feature different roles for the physical and ML model, like e.g. pre-processing.
- Further, different *signals* of physical models are manipulated by ML models. For example, Psichogios and Ungar [9] use an ANN to calculate parameters, whereas Su *et al.* [61] approximate system outputs and Lai *et al.* [82] compute the state derivative.
- A different *number of models* is integrated, the general case is the integration of a single physical model [9], [61], but Thompson and Kramer [77] propose an architecture featuring multiple models (*default* and *output model*).
- The *type* and *abstraction* of models varies, often ANNs are deployed, but whereas e.g. Psichogios and Ungar [9] aim at FFNNs, Su *et al.* [61] use a RNN and Thompson and Kramer [77] employ a RBFN.
- And last but not least, all presented publications provide a dedicated training strategy for the proposed hybrid model structure.

In conclusion, state-of-the-art hybrid modeling captures a variety of industry-relevant combination schemes. However, the original question is missing an important detail that is required to be able to make generally valid statements and thus provide relevance for real engineering applications: It is not discussed in the presented publications if the proposed architectures are meaningful or even solvable in a general, mathematical context. We also realize that it is unacceptable to implement dedicated training methods for different architectures in the industrial field, or at least acknowledge that this significantly slows down the technical progress (e.g. prototyping).

3.2.3 Open challenges

We have justified the need for a more detailed examination and now intend to carry it out. The following two very abstract examples in Fig. 3.11 and 3.12 show, that it is quite easy to build a hybrid model that cannot be simulated. This is also discussed by the author in two publications [83], [84].

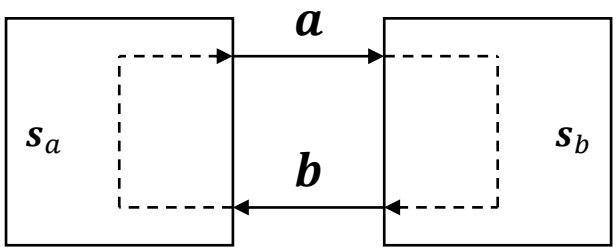


Figure 3.11: Algebraic loops in hybrid modeling: If the input-output-mapping of two connected models $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ can be described by algebraic equations, algebraic loops are constructed, that may be unsolvable or at least need extra attention (identification and tearing).

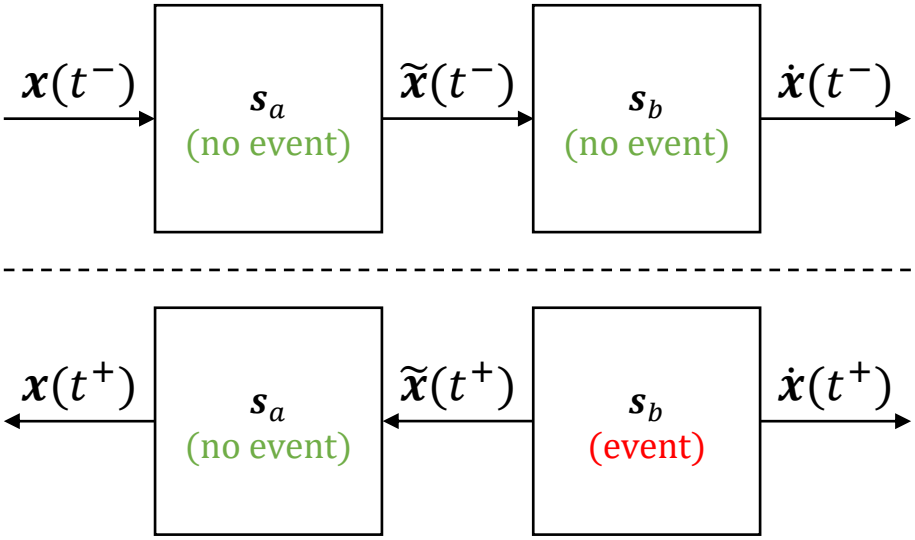


Figure 3.12: Event handling for hybrid models: For systems with events, the causalization of the hybrid model changes during event handling. If no event happens, the left subsystem manipulates the state $\boldsymbol{x}(t^-)$ to $\tilde{\boldsymbol{x}}(t^-)$. The processed state $\tilde{\boldsymbol{x}}(t^-)$ is propagated through the right model to obtain $\dot{\boldsymbol{x}}$. If an event happens, the causalization changes: Inside the right model, the new system state for the right side of the event $\tilde{\boldsymbol{x}}(t^+)$ is computed. This state must be propagated backwards through the left model to obtain $\boldsymbol{x}(t^+)$. This is not considered by state-of-the-art hybrid modeling on *model level*.

As a consequence, the original question should be extended and should read:

How can different models be combined ***meaningful****?*

As shown in Fig. 3.12, this question cannot be answered at the level of causally connected *model blocks*. By investigating acausal models (based on equations) instead of causal models (based on functions), this question can be further discussed. The mathematical structure of the resulting system of equations must be investigated to ensure a meaningful and solvable system. However, to the knowledge of the author, this step was not discussed in the context of generic hybrid modeling yet.

Resulting from the discussion of the state of the art, the following open challenges for hybrid modeling can be summarized:

Solvability The application of design rules, respectively, the combination of blocks, must lead to mathematically meaningful and solvable systems. For example, algebraic loops are often neglected, but can be constructed easily in real applications (compare Fig. 3.11).

Discontinuity Models that are described by or include discontinuous systems require a special event handling routine if integrated as part of hybrid models, because of the changing causalization of the state at event instances (compare Fig. 3.12).

Further challenges, that enhance the quality of a modeling approach, can be derived from the presented hybrid modeling examples:

Interfaces The models inside the presented structures offer varying interfaces, depending on the type of model and abilities. Some physical models, for example, map from a given state $\boldsymbol{x}(t)$ to the next $\boldsymbol{x}(t + \Delta t)$, while others map to the derivative $\dot{\boldsymbol{x}}(t)$ and are paired with a numerical solver. On the other hand, ML models often map from input $\boldsymbol{u}(t)$ to output $\boldsymbol{y}(t)$. Also, some physical models include system inputs $\boldsymbol{u}(t)$ (like Fig. 3.10) and outputs $\boldsymbol{y}(t)$, whereas others do not. Even if this discrepancy is due to the different nature of the models, it drastically complicates the derivation of *actually* general concepts that can easily be applied to a variety of models. For now, if a hybrid modeling scheme is given, models can only be replaced with other models in a straightforward manner if they share the very same interface.

Implementation In general, new hybrid architectures introduce dedicated algorithms for inference/solving and training. This is further investigated in Sec. 3.3.2. In the general case, a new software implementation for every new architecture is needed[5]. Often, code quality and computational performance are not sufficient for applications of industrial scale. Special requirements, such as compatibility with GPUs, are often completely neglected.

Finally, further important challenges are derived from working on the real world use cases presented in Cha. 8, supplemented with experiences and exchanges from multiple research projects[6]:

[5] `GitHub.com` is full of repositories that feature only code for one specific, newly introduced ML structure.

[6] *Unleash Potentials in SIMulation (UPSIM)*, *Proper Hybrid Models for Smarter Vehicles (PHyMoS)* and *Open standards for SCALable virtual engineerING and operation (OpenSCALING)*.

Tooling As stated, classical modeling based on physical principles and machine learning are still two different worlds. As a result, there exist dedicated and specialized tools for both. Doing hybrid modeling, including hybrid model training, in one single tool is not possible now. As a result, some intermediate model must be transferred from one modeling world to the other and vice versa. This requires an adequate exchange format (s. Sec. 7.3) and a new model development workflow (s. Sec. 7.6).

Instability The properties of physical systems, like e.g. energy conservation, are in general well understood by the modeling engineer. This results in conscious and productive further work with the model, for example appropriate numerical solvers can be deployed. If paired with UAs, dynamics are manipulated and the physical properties become unknown and even change over the progress of training. The issues range from suboptimal performance to the complete termination of the simulation. This is further investigated in Sec. 6.3.

Assertions & chattering Simulation models of industrial scale are often developed using assertions to allow for an improved maintainability and extendability of the model. However, these assertions are one of the main sources of errors in hybrid model training, because the incorporated ML model does not consider such assertions. Similarly, the hybrid model is not aware of event indicators, which can lead to unintended triggering of events with high frequency, so-called *chattering*. These issues are further investigated in 6.4 and Sec. 6.5.

Performance Combining different model types (white- and black-box, machine learning and physical), while still maintaining high performance computing requirements is a very challenging venture. However, large parts of this challenge are already studied and captured by the SciML ecosystem of the Julia programming language. Therefore, despite some general aspects of SA (s. Sec. 2.1.3), this thesis will not dive too deep into the topic *computational performance*.

Approval Hybrid models are currently less trustworthy than pure physical models. This is because of the less interpretable nature of the ML part of the model. The release process of hybrid models is for sure a complex topic on its own and will not be further investigated within this work.

Overcome the open challenges

The following sections derive a hybrid modeling paradigm, that addresses the open issues (except *approval*) and therefore provides a versatile foundation for hybrid modeling of industrial scale. With this goal in mind, we pursue the following three steps:

1. In Sec. 3.3, the concept of mixed hybrid universal discrete algebraic ordinary differential equations (HUDA-ODEs) is derived and introduced. This answers the key question on *which* models can be combined as part of hybrid modeling.

2. In Sec. 3.4, HUDA-ODEs are combined in different ways, and we highlight, that the resulting structure remains a solvable HUDA-ODE. This answers the key question on *how* models can be combined (meaningfully).

3. In Sec. 7.3 and further in Sec. A.2 it is shown, that physical models, FMUs, as well as many state of the art ML models can be expressed as HUDA-ODEs, emphasizing the relevance of the derived model class w.r.t. real applications.

These steps result in the following two important consequences that offer a huge benefit over the current state of the art:

1. We can apply methods developed for HUDA-ODEs in Cha. 5 and 6 to a wide variety of physical models and ML models if they belong to the class of HUDA-ODEs.

2. We can easily *combine* physical and/or ML models, the resulting hybrid model still falls in the class of HUDA-ODEs and therefore, of course, derived methods can be used on the combined model as well.

To conclude, these observations allow for the creation of functional, new and powerful hybrid model structures easily while maintaining compatibility with new and existing methods and method implementations.

3.3 Mixed hybrid universal discrete algebraic ordinary differential equations (HUDA-ODEs)

In the following subsections, the HUDA-ODE is introduced and an algorithm for inference is given.

3.3.1 From UDE to HUDA-ODE

In the spirit of hybrid modeling, two well-established concepts — one from engineering and one from SciML itself — merge during the deduction of the HUDA-ODE. From the model engineering side, the thoughts that where developed and collected for over two decades by the FMI design group and resulted in the release of three generations of the FMI standard, are a valuable source and basis for requirements engineering concerning real engineering use cases. On the side of SciML, the concept of UDEs lays not only a mathematical foundation, but also — and probably even more impactful — a *software* foundation for high-end ML by providing the SciML ecosystem, as highlighted in Sec. 2.3.2. The symbiosis of both concepts will benefit from the best of the two worlds:

- FMUs will profit from the ML capabilities and possibilities of UDEs and
- UDEs will profit from the possibility to incorporate simulation models of industrial scale (via FMI).

We start with the point of view of SciML: Rackauckas *et al.* [36] introduce the concept of an UDE as generic forced stochastic delay partial differential equation as follows:

$$\boldsymbol{\mathcal{N}}(\boldsymbol{x}(t), \boldsymbol{x}(\alpha(t)), w(t), \boldsymbol{\upsilon}(\boldsymbol{x}(t), \boldsymbol{p}, \beta(t))) = \mathbf{0}. \tag{3.1}$$

Here, $\boldsymbol{\mathcal{N}}$ is a nonlinear operator, $\boldsymbol{x}$ the system state, $\alpha(t)$ and $\beta(t)$ are delay functions, $w(t)$ is the Wiener process, $\boldsymbol{p}$ are parameters and $\boldsymbol{\upsilon}$ is an universal approximator — hence the name *universal* differential equation. One of the goals of this formalism is to collect requirements for the development of the SciML ecosystem in Julia, which allows for a variety of mathematical models and therefore use cases.

Developing hybrid modeling strategies and methods for the entire set of UDEs is challenging, can lead to suboptimal findings and is unnecessary for many real-world applications. In this thesis, we focus on finding a compromise between generalizability and applicability by deriving a special subset of UDEs, but will show, that a wide variety of engineering use cases can be realized within this set.

Based on the derived requirements for modeling in real life engineering, some mathematical requirements for the proposed model class can be concluded:

- The model class must be a subclass of UDE, to being able to use the performance and possibilities of the SciML ecosystem and
- must be a super class of FMI, to being able to represent models of industrial relevance. This includes allowing for
  - modeling of arbitrary hybrid (discontinuous) ODEs,
  - including algebraic equations,
  - modeling of inputs and outputs (observables), and
  - modeling of discrete systems.

In the following, a concept is derived step by step, that is generic enough to satisfy all proposed mathematical requirements.

Restriction to ODEs

The UDE formalism captures ordinary and partial, as well as stochastic and delay differential equations. Even if all of these equation classes have a right of existence in engineering practice, this thesis especially deals with system simulation in form of ODEs, that can be obtained from DAEs via index reduction or PDEs via appropriate spatial discretization. If only ODEs are considered, the UDE simplifies to an universal ordinary differential equation (UODE), which denotes [36]:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{p}, \boldsymbol{\upsilon}(\boldsymbol{x}(t), \boldsymbol{p}, t), t). \tag{3.2}$$

In the following, we still allow for an actually generic right-hand side, that includes an UA, but don't explicitly put it in as function argument. We pursue naming the resulting structure UODE and denote it by

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{p}, t), \tag{3.3}$$

keeping in mind, that $\boldsymbol{f}$ might be physical equations, a ML model, or a mixture of both. The remaining equation indeed matches the well-known ODE formalism.

Separation of continuous and discrete ODE
To allow for a more intuitive understanding, this concept can be further refined without losing its expressiveness. To further allow for modeling of discrete systems, the system state $\boldsymbol{x}$ can be separated into the *continuous state* $\boldsymbol{x}_c$ and *discrete state* $\boldsymbol{x}_d$, and their dynamics can be defined as

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} \dot{\boldsymbol{x}}_c(t) \\ \dot{\boldsymbol{x}}_d(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}(\boldsymbol{x}_c(t), \boldsymbol{x}_d(t), \boldsymbol{p}, t) \\ \boldsymbol{0} \end{bmatrix}. \tag{3.4}$$

Constant states could be expressed as parameters and are therefore neglected here. We note, that the discrete state is constant over continuous time ($\dot{\boldsymbol{x}}_d = \boldsymbol{0}$), and it can only be changed at an event instance, which fits our definition of the term *discrete* in Sec. 2.2.1.

Super-dense time
For continuous systems, the continuous time can be used to uniquely denote the state $\boldsymbol{x}(t)$ over time. For discontinuous systems, the state changes *immediately* at event time t^*, which results in the circumstance that the continuous time can not be used to uniquely identify the value of the quantity at an event instant. This means, it is not clear if $\boldsymbol{x}(t^*)$ corresponds to the value *before* or *after* event handling. This is further complicated when multiple events are triggered for the same event time t^*.

For this reason, in this work the concept of *super-dense time*, as used in FMI [41], [43] is applied. In super-dense time, the time is defined as a tuple $\boldsymbol{t} = \{t_i, n_i\}$ with $t_i \in \mathbb{R}$ describing the continuous time and $n_i \in \mathbb{N}$ allows for incremental ordering of occurring events. This ordering is defined as [41], [43]:

$$\boldsymbol{t}_1 = \boldsymbol{t}_2 \Leftrightarrow \{t_1, n_1\} = \{t_2, n_2\} \Leftrightarrow t_1 = t_2 \wedge n_1 = n_2 \tag{3.5}$$

and

$$\boldsymbol{t}_1 < \boldsymbol{t}_2 \Leftrightarrow \{t_1, n_1\} < \{t_2, n_2\} \Leftrightarrow t_1 < t_2 \vee (t_1 = t_2 \wedge n_1 < n_2). \tag{3.6}$$

As a result, for the n-th event occurring at a time instant t_i, the point in time before the event can be denoted with $\boldsymbol{t}^- = \{t_i, n-1\}$ and the point after event handling with $\boldsymbol{t}^+ = \{t_i, n\}$. For the state this results in $\boldsymbol{x}(t^-)$ denoting the state before the event, and $\boldsymbol{x}(t^+)$ the state at the same continuous time, but after event handling. In general for time-varying quantities $\phi(t)$, we use $\phi^- \overset{\text{def}}{=} \phi(t^-)$ and $\phi^+ \overset{\text{def}}{=} \phi(t^+)$ for a more compact notation. A deeper introduction to the concept of *super-dense time* can be found in Modelica Association [43].

Extension to hybrid (in terms of discontinuous) systems
To allow for modeling discontinuous systems, the UODE can be extended to the *hybrid* UODE by providing an *event condition* function $\boldsymbol{c}$:

$$\boldsymbol{z}(t) = \boldsymbol{c}(\boldsymbol{x}_c(t), \boldsymbol{x}_d(t), \boldsymbol{p}, t). \tag{3.7}$$

Note, that this is nothing but an additional algebraic equation, complementing the system of equations. Whenever the event indication function $\boldsymbol{c}$ or in short the *event indicator* (terminology as used in [41], [43]) shows a zero crossing, an event is triggered.

In practice, events often depend on either states or time. As a consequence, the event indicator can be separated into a function either depending on the continuous state $\boldsymbol{c}_x$ *or*[7] on time $\boldsymbol{c}_t$:

$$\boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_t(t) \\ \boldsymbol{z}_x(t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{c}_t(\boldsymbol{x}_d(t), \boldsymbol{p}, t) \\ \boldsymbol{c}_x(\boldsymbol{x}_c(t), \boldsymbol{x}_d(t), \boldsymbol{p}) \end{bmatrix}. \tag{3.8}$$

This measure leads to further computational benefits, as briefly shown in the following. The next state-dependent event needs to be determined during solving the ODE by monitoring event indicators that depend on the continuous state and therefore change with every step of the ODE solver. The time-dependent event, on the other hand, must only be examined at event instances, because it does not depend on the changing continuous state, but only on the discrete state (that does not change between events) and the parameters (that are constant). Because $\boldsymbol{c}_t$ only depends on a single continuous unknown — the time — in practice, e.g. in FMI, the function argument t is extracted. For any event time t^-, at least one entry of the event condition $\boldsymbol{z}_t$ is zero before event-handling:

$$\boldsymbol{z}_t(t^-) = \boldsymbol{c}_t(\boldsymbol{x}_d(t^-), \boldsymbol{p}, t^-) \Rightarrow \exists i \in \{1, ..., |\boldsymbol{z}|\} \quad \text{such that} \quad z_{ti}(t^-) = 0. \tag{3.9}$$

We can rearrange $\boldsymbol{c}_t$ into $\boldsymbol{c}_{t^*}$ by solving for all event time instances:

$$\boldsymbol{t}^* = \boldsymbol{c}_{t^*}(\boldsymbol{x}_d, \boldsymbol{p}), \tag{3.10}$$

where $\boldsymbol{t}^*$ is a vector holding all event times for pure time-dependent events. Doing so, the next event time — only for purely time-dependent events — can be requested by the ODE solver by evaluation of $\boldsymbol{c}_{t^*}$ based on the discrete state and parameterization.

If an event is detected — independent of it depends on state-, time-, or state-time — the event affect function $\boldsymbol{a}$ determines a new state $\boldsymbol{x}(t^+)$ for the system:

$$\boldsymbol{x}(t^+) = \boldsymbol{a}(\boldsymbol{x}(t^-), \boldsymbol{p}, t^-, \boldsymbol{q}^-). \tag{3.11}$$

As can be seen, the event affect function does not explicitly include the triggered event index, but a binary vector $\boldsymbol{q}^-$ identifying *all* indicators that show zero crossings *before* event handling. This is because multiple events can happen simultaneously, and the execution order is not defined in this case. By providing the set of zero crossings via $\boldsymbol{q}$, the decision on how to handle simultaneous events is shifted from the level of *event handling* to the level of *modeling*.

Extension by algebraic (output) equations

In addition to states, it is often meaningful to define further quantities, referred to as *outputs* in systems and control theory. These outputs depend solely and algebraically on the system state, parameters, time, and input (introduced in the next paragraph). The system of equations can be extended by an additional set of algebraic equations:

$$\boldsymbol{y}(t) = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{p}, t). \tag{3.12}$$

[7] However, if state-and-time-dependent events are required, the continuous event indicator can be extended to further allow for the time t.

Strictly speaking, the output could also depend on the state derivative $\dot{\boldsymbol{x}}$, but this is not explicitly included as the state derivative can also be calculated via the arguments $\boldsymbol{x}$, $\boldsymbol{p}$ and t. It is also important to clarify, that the resulting system is still an ODE because the algebraic variable $\boldsymbol{y}(t)$ is *calculated* based on the state, parameters and time and can therefore not act as *algebraic constraint*.

Extension by inputs
System inputs are usually available in discretized form (e.g. data points). For example, signals from real sensors are often tried to be recorded equidistantly (with a fixed frequency), which is not possible *exactly* in practice. There are multiple ways to deal with a time series of (non-equidistant) input signals:

(a) The input points over time can be interpolated (e.g. by a piecewise interpolation polynomial) or approximated (e.g. polynomial regression) and added as symbolic function to the system of equations. This is further referred to as *continuous input*. Even if this is straightforward to implement, the dynamics of the input signals are not captured by the numerical solver if the input does not affect the state derivative, and step size control does not guarantee appropriate sampling of the input signal in such a case. Technically, *continuous inputs* can be implemented by extending the system of equations by a new function $\boldsymbol{u}(t)$, which further allows for the exchange of the input signal and makes the system explicitly time-dependent.

(b) If the input derivative is known analytically or can be approximated, the input signal can be added as additional continuous state by providing its derivative function. Because of the increasing discrepancy between numerical solution (integrated input derivative) and the recorded input over time, events should be added[8] to synchronize the input state (that is determined by numerical integration of the provided derivative function) with the recorded data points. Besides the extended state space, this introduces additional time events, but allows for capturing the input signal dynamics by the step size control of the solver. This is further referred to as *continuous state inputs*.

(c) The discrete time points where the input signal is captured can be interpreted as time events. All inputs are modeled as discrete states. During event handling, the corresponding discrete state is updated with the value of the recorded input. This is further referred to as *discrete state input*. This way, no changes in the input signals are missed by the numerical solver, but the input is assumed a flat line between measurement points (*sample-and-hold*).

(d) For the sake of completeness, also constant inputs are possible, but should be handled as parameters in this case, which is referred to as *parameter inputs*.

To conclude, depending on the requirements of the use case, inputs can be modeled as external function $\boldsymbol{u}$ or as part of the right-hand side (continuous input), as part of the discrete state and event functions (discrete state input), the continuous

[8]These could be time events that synchronize the input state at specific points in time (e.g. periodically) or state events, that synchronize the input as soon as the deviation becomes to large. This can be implemented by providing an event indicator comparing the input state to data.

state and event functions (continuous state input) or the parameter vector (parameter input). The approaches are exemplified in Fig. 3.13, except for the constant parameter input because of its triviality.

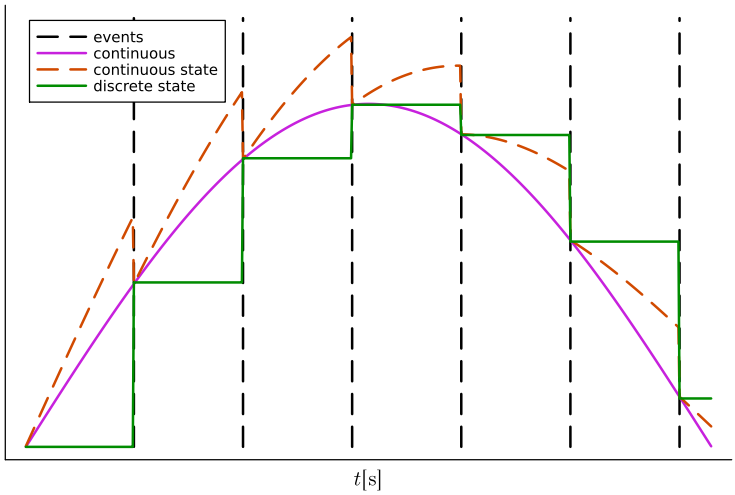


Figure 3.13: Comparison of continuous, continuous state and discrete state inputs.

Note, that by allowing $\boldsymbol{u}(t)$ being dependent on the system state $\boldsymbol{x}$ (or output $\boldsymbol{y}$, if it depends on the state), a *controlled* system is derived. In the following, additional functions can be defined optionally, that improve the practical usability of the HUDA-ODE.

Optional: Error indication
During modeling of physical systems, often assertions are defined. For example, parameters (e.g. mass must always be real positive) or states (e.g. angles close to singularity) need to be in specified ranges. Such an error function can be optionally added to the HUDA-ODE formalism:

$$\boldsymbol{e}(t) = \boldsymbol{\delta}(\boldsymbol{x}(t), \boldsymbol{u}, \boldsymbol{p}, t). \tag{3.13}$$

This can not only being used to check for the violation of e.g. model assumptions, but also to actively prevent them if this function is differentiable, as shown in Sec. 6.5.

Optional: Initialization and termination
For solving ODEs, an initial state $\boldsymbol{x}_0$ must be provided. In some applications, the initial state is not (completely) known and is determined by providing other variables like inputs, outputs or parts of the state. This is referred to solving the *initialization problem*:

$$\boldsymbol{x}_0 = \boldsymbol{\alpha}(..., t_0). \tag{3.14}$$

The initialization problem does not need to have a unique solution. Initialization of hybrid (in terms of *discontinuous*) systems is explained in more detail in Ochel and Bachmann [85]. Similar to initialization, an optional method for completing the solution can be defined. However, this function is only of technical relevance (e.g. to release memory), does not influence the shape of the solution trajectory and is not considered further in terms of methodology.

To conclude, please note that the HUDA-ODE is nothing but a different organized view of the UODE, in terms of a more strict separation of different equation types (algebraic, ordinary differential and discrete) and variability (continuous, discrete, constant). Both concepts can be transferred into each other from a mathematical point of view. However, the following strategies — especially rules for combination and event handling — require the shape of a HUDA-ODE to be explained and implemented more efficiently. This is because by the more detailed separation we can deal with different parts of the system of equations differently and more intuitively.

HUDA-ODE at a glance

To conclude, the resulting HUDA-ODE almost matches the simulation model definition of the FMI standard [41], [43], which was used as a template as introduced. The major differences are:

- The right-hand side $\boldsymbol{f}$, the output function $\boldsymbol{g}$, the event condition $\boldsymbol{c}$ and the event affect $\boldsymbol{a}$ explicitly combine physical models *and/or* UAs. Even if this is not prohibited by FMI, it is not explicitly supported and is not part of the core scope. For example, FMI does not enforce availability of parameter sensitivities that are crucial for training with large parameter spaces.
- The discrete state $\boldsymbol{x}_d$ is known for HUDA-ODEs, whereas it is in general inaccessible[9] in FMI. Complications resulting from this restriction are discussed in Sec. 7.3. Further, FMUs only return the continuous part of the state $\boldsymbol{x}_c^+$ at event instances after event handling, instead of the entire new state $\boldsymbol{x}^+$.
- The event affect function of FMI does not need identification (via argument $\boldsymbol{q}$) of the event indicators that triggered the event to be handled. The event indicator zero crossings are stored internally as part of the FMU.
- The optional error indication function $\boldsymbol{\delta}$ is not part of FMI, however, would allow for an optimized training of hybrid models involving FMUs (s. Sec. 6.5).

Putting the system of equations together results in:

$$\begin{bmatrix} \dot{\boldsymbol{x}}_c(t) \\ \dot{\boldsymbol{x}}_d(t) \\ \boldsymbol{y}(t) \\ \boldsymbol{z}(t) \\ \boldsymbol{x}(t^+) \end{bmatrix} = \begin{bmatrix} \boldsymbol{f}(\boldsymbol{x}_c(t), \boldsymbol{x}_d(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \\ \boldsymbol{0} \\ \boldsymbol{g}(\boldsymbol{x}_c(t), \boldsymbol{x}_d(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \\ \boldsymbol{c}(\boldsymbol{x}_c(t), \boldsymbol{x}_d(t), \boldsymbol{u}(t), \boldsymbol{p}, t) \\ \boldsymbol{a}(\boldsymbol{x}_c(t^-), \boldsymbol{x}_d(t^-), \boldsymbol{u}(t^-), \boldsymbol{p}, t^-, i) \end{bmatrix}. \tag{3.15}$$

It is worth mentioning, that these five equation blocks are completely independent in the sense that they do not depend on intermediate results from each other.

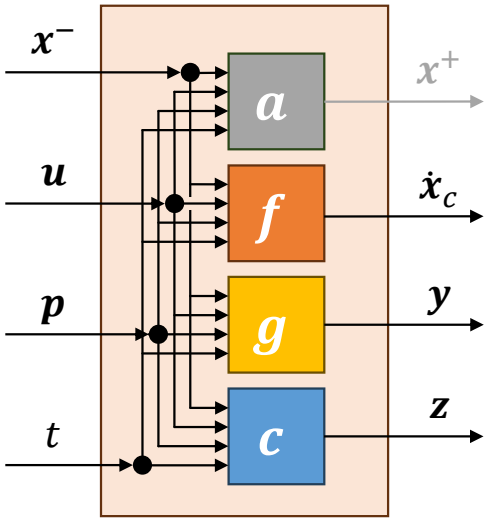
Contribution 1 *The HUDA-ODE is formally derived and introduced, that offers a different view of the UODE, driven by requirements from real world engineering applications (gathered from FMI). This way, HUDA-ODEs offer a common interface for solving and training for a variety of industrial and research use cases. Both concepts, HUDA-ODEs and UODEs, can be transformed*

[9] However, some tools like *Dymola* (Dassault Systèmes) use special identifiers to identify discrete states. This information can be used if present.

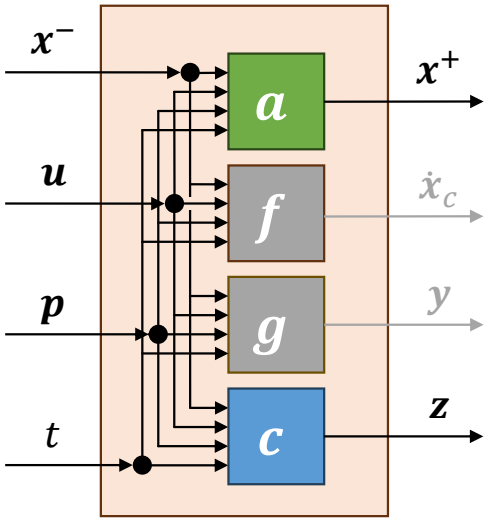
into each other. As a consequence, methods for HUDA-ODEs are applicable for UODEs as well and vice versa. However, HUDA-ODE provide — next to the ODE interface — a dedicated interface for event handling, discrete systems and further functionality (initialization, termination, errors) which allows for a target-oriented discussion of hybrid modeling involving such models.

3.3.2 Solving of HUDA-ODEs

Mixed continuous-discrete systems need to be solved in an alternating way. During the *continuous phase*, the state is passed on to the HUDA-ODE and a state derivative is obtained, that is numerically integrated over time by the ODE solver (compare Fig. 3.14a). This way, time proceeds and a new state is obtained. If a discontinuous state change is needed, which is identified by a zero crossing in one or more of the event indicators, the continuous phase is left, and the *discrete phase* is entered (compare Fig. 3.14b). During the discrete system simulation, a new system state (continuous and/or discrete part) is calculated. It is important to note, that during the discrete phase no progress in continuous time is made.



(a) **Continuous simulation**
During continuous system simulation, the continuous system state $\boldsymbol{x}_c^-$ is determined by the numerical solver and the state $\boldsymbol{x}^-$ is input to the HUDA-ODE. Based on the function $\boldsymbol{f}$, the continuous state derivative $\dot{\boldsymbol{x}}_c$ is computed and returned to the solver. Further, the functions for outputs $\boldsymbol{g}$ and event indicators (to check for events) $\boldsymbol{c}$ are evaluated.



(b) **Discrete simulation (events)**
During discrete system simulation, the entire system state $\boldsymbol{x}^+$ is determined by evaluating the function $\boldsymbol{a}$ of the HUDA-ODE. The continuous part of the new state $\boldsymbol{x}_c^+$ is input to the numerical solver. Because new states could trigger new events, the condition $\boldsymbol{c}$ is evaluated after every handled event instance.

Figure 3.14: The solving of a HUDA-ODE, separated into the continuous and discrete subsystem simulation.

The solution procedure for HUDA-ODEs (motivated by [41], [43]) can be formally summarized by Alg. 1. The inputs $\boldsymbol{u}$ are omitted for readability, but the algorithm can be extended straightforward by evaluating $\boldsymbol{u}$ within every function call to $\boldsymbol{\alpha}$, $\boldsymbol{f}$,

$\boldsymbol{g}$, $\boldsymbol{c}$, $\boldsymbol{a}$ and $\boldsymbol{\omega}$.

Algorithm 1 Solving algorithm for HUDA-ODEs.

Input: $\boldsymbol{\tau} := \{t_0, ..., t_n\}$ ▷ $n+1$ time points to save the solution at
$i := 0$ ▷ start with step zero
$\boldsymbol{x}_{ci}, \boldsymbol{x}_{di} := \boldsymbol{\alpha}(...)$ ▷ initialize state
$\boldsymbol{y}_i := \boldsymbol{g}(\boldsymbol{x}_{ci}, \boldsymbol{x}_{di}, \boldsymbol{p}, t_i)$ ▷ initial outputs
$\boldsymbol{z} := \boldsymbol{c}(\boldsymbol{x}_{ci}, \boldsymbol{x}_{di}, \boldsymbol{p}, t_i)$ ▷ initialize event indicators
while $i < |\boldsymbol{\tau}|$ **do** ▷ as long not all steps done
  $\boldsymbol{x}_{di+1} := \boldsymbol{x}_{di}$ ▷ assume that the discrete state does not change
  $t^* := t_i$ ▷ initialize event time variable to enter loop
  while $t^* < t_{i+1}$ **do** ▷ as long as next time point not reached
    $\boldsymbol{x}_{ci+1}, t^* := ODESolveEvent(\boldsymbol{f}, \boldsymbol{c}, \boldsymbol{x}_{ci}, \boldsymbol{x}_{di}, \boldsymbol{p}, t_{i+1})$ ▷ solve next step
    $\boldsymbol{z}_{new} := \boldsymbol{c}(\boldsymbol{x}_{ci+1}, \boldsymbol{x}_{di+1}, \boldsymbol{p}, t^*)$ ▷ retrieve event indicators
    $\boldsymbol{q} := cross(\boldsymbol{z}, \boldsymbol{z}_{new})$ ▷ check for switching signs
    while $\|\boldsymbol{q}\| \geq 1$ **do** ▷ at least one zero crossing, so an event happening
      $\boldsymbol{x}_{ci+1}, \boldsymbol{x}_{di+1} := \boldsymbol{a}(\boldsymbol{x}_{ci+1}, \boldsymbol{x}_{di+1}, \boldsymbol{p}, t^*, \boldsymbol{q})$ ▷ handle events
      $\boldsymbol{z} := \boldsymbol{z}_{new}$ ▷ overwrite cached event indicators
      $\boldsymbol{z}_{new} := \boldsymbol{c}(\boldsymbol{x}_{ci+1}, \boldsymbol{x}_{di+1}, \boldsymbol{p}, t^*)$ ▷ retrieve event indicators
      $\boldsymbol{q} := cross(\boldsymbol{z}, \boldsymbol{z}_{new})$ ▷ check for switching signs
    end while ▷ check if event triggered another event
  end while ▷ check if t_{i+1} reached
  $\boldsymbol{y}_{i+1} := \boldsymbol{g}(\boldsymbol{x}_{ci+1}, \boldsymbol{x}_{di+1}, \boldsymbol{p}, t_{i+1})$ ▷ evaluate outputs
  $i := i + 1$ ▷ increment step count
end while ▷ check if $|\boldsymbol{\tau}|$ reached
$\omega(...)$ ▷ clean-up, finalize
Output: $\boldsymbol{X} = \{\boldsymbol{x}_0, ..., \boldsymbol{x}_i\}$ ▷ return state values
Output: $\boldsymbol{Y} = \{\boldsymbol{y}_0, ..., \boldsymbol{y}_i\}$ ▷ return output values

Here, the algorithm *cross* is defined as

$$\boldsymbol{q} = cross(\boldsymbol{z}, \boldsymbol{z}^*) = \begin{cases} q_i = 1 & \text{if } z_i > 0 \wedge z_i^* < 0 \\ q_i = 1 & \text{if } z_i < 0 \wedge z_i^* > 0 \\ q_i = 0 & \text{else} \end{cases} \quad \text{for } i \in 1, \cdots, |\boldsymbol{z}|. \tag{3.16}$$

Note, that within this definition, only real zero crossings (from positive to negative and vice versa) are interpreted as events, which explicitly excludes the case of an event indicator "touching" the zero line without explicitly crossing it.

Further, *ODESolveEvent* is slightly modified to take $\boldsymbol{c}$ (the vectorial zero crossing function) instead of c_e (the scalar zero crossing function) and the entire system state $\boldsymbol{x}_c$ and $\boldsymbol{x}_d$ instead of only the continuous state $\boldsymbol{x}_c$ as input arguments.

Based on the applied requirements for deriving HUDA-ODEs, it is clear that various structures such as FMUs (with minor limitations, see Sec. 7.3), different types of ANNs, as well as NODEs automatically fall into this class. Rediscovering such known concepts in HUDA-ODEs is part of the appendix Sec. A.2.

3.4 Combination of HUDA-ODEs

After defining a class of models for hybrid modeling together with an algorithm for solving, the combination of models of this class is investigated. In this section we want to prove that the combination of two (or more) HUDA-ODEs leads to a solvable HUDA-ODE again. This provides the foundation for modeling arbitrary hybrids, consisting of ML and/or physical models, and apply the methods presented in Cha. 5 and 6 to them. This section is also discussed by Thummerer and Mikelsons [84] in less detail.

3.4.1 Generic architecture

In the following, we derive the *generic* combination of two HUDA-ODEs and prove that this leads to a HUDA-ODE again. For this, we make four observations:

1. The event and error indicators, affect, and output equations are algebraic equations. Even the *ordinary differential equation* part of the system can just be handled the same as an "algebraic" equation, with the special case that the variable *state* is determined by numerical integration of the *state derivative*. However, we can ignore this connection for a moment and treat both variables individually. From this point of view, the HUDA-ODE consists only of "algebraic" equations.

2. It is sufficient to examine systems with vectorial inputs (*knowns*) $\boldsymbol{\upsilon}$ and outputs (*unknowns*) $\boldsymbol{\gamma}$, because multiple vectorial inputs/outputs can be concatenated to a single vectorial input/output again. For example, a system with two vectorial inputs $\boldsymbol{x}$, $\boldsymbol{u}$ and one scalar input t, can be expressed with a single input vector that states $\boldsymbol{\upsilon} = [x_1, \cdots, x_n, u_1, \cdots, u_m, t]^T$ for $n = |\boldsymbol{x}|$ and $m = |\boldsymbol{u}|$.

3. Connecting unknowns (derivatives, outputs, event and error indicators) with knowns (states, inputs, parameters) of another HUDA-ODE results in chaining the algebraic equations. This is not limited from a theoretical perspective, but algebraic loops must be considered. However, inputs cannot be connected to inputs or outputs to outputs, because this would destroy the causality of the system of equations.

4. We observe, that the event condition and affect functions are defined *locally*. This means that state changes at discontinuities are defined for the *local* subsystem and must therefore be propagated through the other subsystems around to obtain the new *global* state for the solver. This is further investigated in Sec. 3.4.4.

Based on these observations, the combination of two HUDA-ODEs can mathematically be discussed as the combination of two (or more) systems of algebraic equations. Two algebraic systems $\boldsymbol{\gamma}_a = \boldsymbol{s}_a(\boldsymbol{\upsilon}_a)$ and $\boldsymbol{\gamma}_b = \boldsymbol{s}_b(\boldsymbol{\upsilon}_b)$ can be combined to a new system (*combined model*) $\boldsymbol{\gamma}_z = \boldsymbol{s}_z(\boldsymbol{\upsilon}_z)$, as can be seen in Fig. 3.15. However, this introduces three new vectorial unknowns $\boldsymbol{\upsilon}_a$, $\boldsymbol{\upsilon}_b$ and $\boldsymbol{\gamma}_z$.

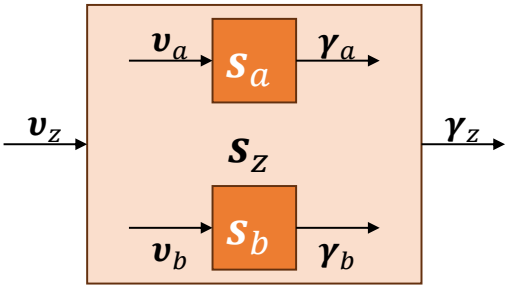


Figure 3.15: Two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ can be combined to a single one $\boldsymbol{s}_z$. Because there are new unknowns $\boldsymbol{v}_a$, $\boldsymbol{v}_b$, $\boldsymbol{\gamma}_z$ introduced, the system is not well-posed. This can be seen from the missing connections between the outer input $\boldsymbol{v}_z$ and outer output $\boldsymbol{\gamma}_z$ and the inner inputs $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$ and outputs $\boldsymbol{\gamma}_a$ and $\boldsymbol{\gamma}_b$.

This combination of systems is sometimes referred to as *model integration*, but this terminology is not used further because it can easily be confused with the *numerical integration* of an ODE. From a mathematical (vectorial) perspective, this is equivalent to the concatenation of both systems of equations $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$:

$$\begin{bmatrix} \boldsymbol{\gamma}_a \\ \boldsymbol{\gamma}_b \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_a(\boldsymbol{v}_a) \\ \boldsymbol{s}_b(\boldsymbol{v}_b) \end{bmatrix}. \tag{3.17}$$

It is worth mentioning, that both algebraic systems are completely independent for now so they do not share any variables. As stated, three new unknowns $\boldsymbol{v}_a$, $\boldsymbol{v}_b$, $\boldsymbol{\gamma}_z$ are introduced. We define three linear equations $\boldsymbol{c}_a, \boldsymbol{c}_b, \boldsymbol{c}_z$ in order to fix the missing connections, which we define as follows:

$$\boldsymbol{v}_a = \boldsymbol{c}_a(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{v}_z) = \boldsymbol{W}_{aa}\boldsymbol{\gamma}_a + \boldsymbol{W}_{ab}\boldsymbol{\gamma}_b + \boldsymbol{W}_{az}\boldsymbol{v}_z \qquad (+\,\boldsymbol{b}_a) \tag{3.18}$$

$$\boldsymbol{v}_b = \boldsymbol{c}_b(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{v}_z) = \boldsymbol{W}_{ba}\boldsymbol{\gamma}_a + \boldsymbol{W}_{bb}\boldsymbol{\gamma}_b + \boldsymbol{W}_{bz}\boldsymbol{v}_z \qquad (+\,\boldsymbol{b}_b) \tag{3.19}$$

$$\boldsymbol{\gamma}_z = \boldsymbol{c}_z(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{v}_z) = \boldsymbol{W}_{za}\boldsymbol{\gamma}_a + \boldsymbol{W}_{zb}\boldsymbol{\gamma}_b + \boldsymbol{W}_{zz}\boldsymbol{v}_z \qquad (+\,\boldsymbol{b}_z), \tag{3.20}$$

with connection matrix $\boldsymbol{W} \in \mathbb{R}^{n\times m}$ and connection bias $\boldsymbol{b} \in \mathbb{R}^n$ with $n = |\boldsymbol{v}_a| + |\boldsymbol{v}_b| + |\boldsymbol{\gamma}_z|$ and $m = |\boldsymbol{\gamma}_a| + |\boldsymbol{\gamma}_b| + |\boldsymbol{v}_z|$:

$$\boldsymbol{W} = \begin{matrix} \\ \boldsymbol{v}_a \\ \boldsymbol{v}_b \\ \boldsymbol{\gamma}_z \end{matrix} \begin{matrix} \begin{matrix} \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_b & \boldsymbol{v}_z \end{matrix} \\ \begin{bmatrix} \boldsymbol{W}_{aa} & \boldsymbol{W}_{ab} & \boldsymbol{W}_{az} \\ \boldsymbol{W}_{ba} & \boldsymbol{W}_{bb} & \boldsymbol{W}_{bz} \\ \boldsymbol{W}_{za} & \boldsymbol{W}_{zb} & \boldsymbol{W}_{zz} \end{bmatrix} \end{matrix} \qquad \boldsymbol{b} = \begin{matrix} \boldsymbol{v}_a \\ \boldsymbol{v}_b \\ \boldsymbol{\gamma}_z \end{matrix} \begin{bmatrix} \boldsymbol{b}_a \\ \boldsymbol{b}_b \\ \boldsymbol{b}_z \end{bmatrix}. \tag{3.21}$$

The connection matrix $\boldsymbol{W}$ can be interpreted and used in two different ways:

- The entries of the connection submatrices can be populated with 0 or 1 to indicate the existence (1) or non-existence (0) of a connection between the two corresponding variables.
- The entries can be initialized (e.g. with 0 or 1) and be optimized together with physical and/or ML parameters. This is done by extending the parameter vector of the HUDA-ODE. The resulting structure is capable of describing arbitrary connections between two given algebraic systems (or HUDA-ODEs) respectively.

The complete system including the new connection equations now states:

$$\begin{bmatrix} \boldsymbol{\gamma}_a \\ \boldsymbol{\gamma}_b \\ \boldsymbol{v}_a \\ \boldsymbol{v}_b \\ \boldsymbol{\gamma}_z \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_a(\boldsymbol{v}_a) \\ \boldsymbol{s}_b(\boldsymbol{v}_b) \\ \boldsymbol{c}_a(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{v}_z) \\ \boldsymbol{c}_b(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{v}_z) \\ \boldsymbol{c}_z(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{v}_z) \end{bmatrix}. \tag{3.22}$$

For a more intuitive understanding, the system can further be visualized, see Fig. 3.16. Here, every arrow corresponds to a part of the connection matrix $\boldsymbol{W}$.

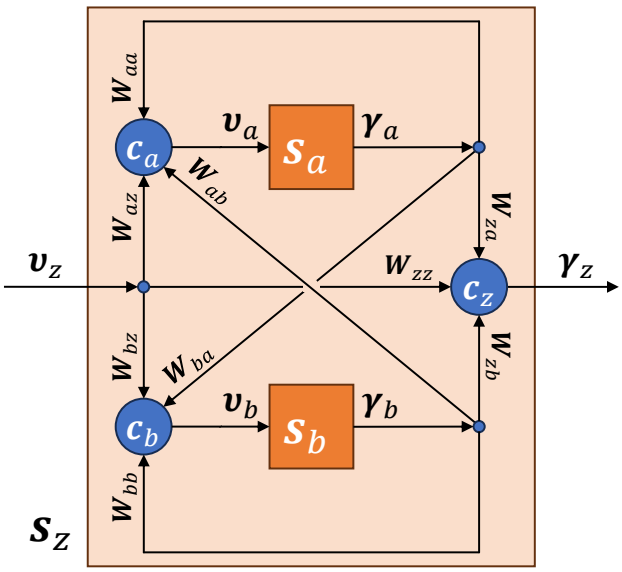


Figure 3.16: The generic combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ (orange squares), containing nested sets of algebraic loops. For example, the subsystem $\boldsymbol{s}_b$ is evaluated based on the intermediate results of $\boldsymbol{s}_a$, and the result is fed back into $\boldsymbol{s}_a$ again. Further, both subsystems form algebraic loops with themselves, because their outputs are connected to their own inputs. A direct feed through (DFT) between $\boldsymbol{v}_z$ and $\boldsymbol{\gamma}_z$ is present. The connection equations $\boldsymbol{c}_a, \boldsymbol{c}_b$ and $\boldsymbol{c}_z$ are shown as blue circles, and each connection arrow corresponds to a part of the connection matrix $\boldsymbol{W}$.

3.4.2 Interpretability

As already introduced, the connections expressed by the matrix $\boldsymbol{W}$ and bias $\boldsymbol{b}$ can either be *specified* (by the modeler) or *identified* (by optimization). The following different *signal operations* can be represented by the linear connection equations $\boldsymbol{c}_a$, $\boldsymbol{c}_b$ and $\boldsymbol{c}_z$, depending on the chosen dimensions for the combined system input $\boldsymbol{v}_z$ and output $\boldsymbol{\gamma}_z$:

Connection As already stated, connections in $\boldsymbol{W}$ can be established ($= 1$) or not ($= 0$).

Gain Further, signals in $\boldsymbol{W}$ can be amplified (> 1) or attenuated ($< 1 \wedge > 0$) or even negated (< 0).

Shift For $\boldsymbol{b} \neq \boldsymbol{0}$, signals can be shifted to a higher or lower milieu.

Mixing If $\boldsymbol{W}$ hast the shape of a permutation matrix (exactly one element of $\boldsymbol{W}$ per row/column is not zero), signals are connected in a one-to-one arrangement. If multiple elements per row are occupied, the corresponding signals are (linearly) mixed from multiple input signals.

Cloning For $|\boldsymbol{\upsilon}_z| = |\boldsymbol{\upsilon}_a| = |\boldsymbol{\upsilon}_b|$ and $\boldsymbol{W}_{az} = \boldsymbol{W}_{bz}$, both subsystems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ get exactly the same signals as inputs.

Addition Similarly, for $|\boldsymbol{\gamma}_z| = |\boldsymbol{\gamma}_a| = |\boldsymbol{\gamma}_b|$ and $\boldsymbol{W}_{za} = \boldsymbol{W}_{zb}$, the outputs of both subsystems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ are added to obtain the combined system output.

Splitting For $|\boldsymbol{\upsilon}_z| = |\boldsymbol{\upsilon}_a| + |\boldsymbol{\upsilon}_b|$ and the vertical concatenation of $\begin{bmatrix}\boldsymbol{W}_{az}\\ \boldsymbol{W}_{bz}\end{bmatrix}$ is a permutation matrix, the system input is divided disjointly between the two subsystems — so no signals are shared.

Concatenation Similarly, for $|\boldsymbol{\gamma}_z| = |\boldsymbol{\gamma}_a| + |\boldsymbol{\gamma}_b|$ and the horizontal concatenation $\begin{bmatrix}\boldsymbol{W}_{za} & \boldsymbol{W}_{zb}\end{bmatrix}$ is a permutation matrix, the combined system output is composed of the individual outputs from both subsystems, without mixing them.

Further, these connections can be visualized by plotting parts of the connection matrix $\boldsymbol{W}$, as shown in the corresponding experiment.

3.4.3 Algebraic loops

For discussing the solvability of a system of equations concerning the existence of algebraic loops, the dependency matrix $\boldsymbol{D}$ can be examined:

$$\boldsymbol{D} = \begin{array}{c} \\ \boldsymbol{s}_a \\ \boldsymbol{s}_b \\ \boldsymbol{c}_a \\ \boldsymbol{c}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_b & \boldsymbol{\upsilon}_a & \boldsymbol{\upsilon}_b & \boldsymbol{\gamma}_z \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{D}_a & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{D}_b & \boldsymbol{0} \\ \boldsymbol{D}_{aa} & \boldsymbol{D}_{ab} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}_{ba} & \boldsymbol{D}_{bb} & \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} \\ \boldsymbol{D}_{za} & \boldsymbol{D}_{zb} & \boldsymbol{0} & \boldsymbol{0} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array}. \tag{3.23}$$

This matrix is sometimes referred to as *structure incidence matrix* in literature concerning object-oriented modeling, however, we deliberately avoid using this term here, since the matrix shown does not formally satisfy the definition of an incidence matrix in graph theory[10]. Let $\boldsymbol{J} \in \mathbb{R}^{n\times m}$ denote the Jacobian of the equations (rows) with respect to the variables (columns):

$$\boldsymbol{J} = \frac{\partial[\boldsymbol{s}_a\ \boldsymbol{s}_b\ \boldsymbol{c}_a\ \boldsymbol{c}_b\ \boldsymbol{c}_z]^T}{\partial[\boldsymbol{\gamma}_a\ \boldsymbol{\gamma}_b\ \boldsymbol{\upsilon}_a\ \boldsymbol{\upsilon}_b\ \boldsymbol{\upsilon}_z]^T}, \tag{3.24}$$

where $n = |\boldsymbol{s}_a| + |\boldsymbol{s}_b| + |\boldsymbol{c}_a| + |\boldsymbol{c}_b| + |\boldsymbol{c}_z|$ and $m = |\boldsymbol{\gamma}_a| + |\boldsymbol{\gamma}_b| + |\boldsymbol{\upsilon}_a| + |\boldsymbol{\upsilon}_b| + |\boldsymbol{\upsilon}_z|$. The dependency matrix $\boldsymbol{D} \in \mathbb{R}^{n\times m}$ can then be interpreted as simplification of the Jacobian $\boldsymbol{J}$, in a sense that the entries are either zero for zero entries, or one for non-zero entries, formally

$$D_{i,j} = \begin{cases} 0 & \text{for } J_{i,j} = 0 \\ 1 & \text{elsewhere} \end{cases} \tag{3.25}$$

[10] Exactly two non-zero entries per column for undirected graphs.

The dependency matrices for the connection block matrices $\boldsymbol{W}_{xy}$ with $x, y \in \{a, b, z\}$ are denoted with $\boldsymbol{D}_{xy}$. Although they are related[11], the dependency matrix $\boldsymbol{D}$ does not contain all the values of the matrix $\boldsymbol{W}$, as the last column of $\boldsymbol{W}$ describes the connections with the input of the combined model $\boldsymbol{v}_z$, which is known and therefore by definition not part of a dependency matrix. The dependency matrices of the two subsystems $\boldsymbol{D}_a$ and $\boldsymbol{D}_b$ are also included. For non-trivial systems, it states that $\boldsymbol{D}_a \neq \boldsymbol{0}$ and $\boldsymbol{D}_b \neq \boldsymbol{0}$, which leads to a combined system with algebraic loops (compare Eq. 3.23). This can be formally verified by a lower triangular (LT) transformation, which is obviously not feasible, since all rows of the matrix $\boldsymbol{D}$ have at least two non-zero entries.

Theoretically, algebraic loops are unproblematic, as they can be identified (e.g. Tarjan's algorithm [86]), reduced (e.g. *Reshuffling* [87]), teared, and solved (e.g. Newton's method) using established methods. In practice, the solving of algebraic loops may be expensive, and it is not guaranteed that a solution can be found — or even exists. Further, the process for identification is associated with a high computational effort besides the algorithm complexity, as this step must be repeated after each re-parameterization as new connections might be created or old ones are deleted. For the training of hybrid models, this means these steps must be repeated after every training step or parameter adaption respectively. At the time of writing, there was no use case in progress that justified the construction (and therefore monitoring) of additional algebraic loops by the combination of two systems. Because of the large computational overhead for the presented generic architecture, online identification and tearing of algebraic loops, as well as the handling of discontinuities, is not further investigated for the *general* combination architecture. Instead, a more efficient architecture is derived in the following by slightly restricting parameters of the matrix $\boldsymbol{W}$. Doing so, new algebraic loops by connections can be excluded.

PSD: Parallel-sequential-DFT

Based on the proposed *generic* combination model that might feature additional algebraic loops, it seems logic to derive the most generic model *without* any additional algebraic loops by design — caused by connecting submodels. This combination structure is highlighted in the following.

For deriving such an architecture, we make the following decisions: First, connections of the subsystems with themselves are prohibited by enforcing $\boldsymbol{W}_{aa} = \boldsymbol{0}$ and $\boldsymbol{W}_{bb} = \boldsymbol{0}$. The direct feed through (DFT), on the other hand, causes no algebraic loops independent of the parameterization and is therefore maintained. Further, the parallel structure of $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ is fully preserved, because subsystems evaluated in parallel are fully separated and are therefore not critical. However, the sequential connections are restricted to the case that $\boldsymbol{s}_a$ is evaluated *before* $\boldsymbol{s}_b$ or $\boldsymbol{s}_b$ is evaluated *before* $\boldsymbol{s}_a$. As this prevents the unhindered execution of the two subsystems one after the other more than a single time, it is already understandable that no more algebraic loops occur. This is shown formally below. The resulting architecture is referred to as *PSD* (parallel, sequential and DFT). To formally summarize this derivation, for the *PSD* architecture it states that $\boldsymbol{W}_{aa} = \boldsymbol{0}$ and $\boldsymbol{W}_{bb} = \boldsymbol{0}$. Further, two cases can be distinguished:

PSDa Subsystem $\boldsymbol{s}_a$ is evaluated first, yielding $\boldsymbol{W}_{ab} = \boldsymbol{0}$.

[11] Note, that the partial derivative of a linear function w.r.t. the unknown variable is the linear coefficient of the equation, in this case exactly the value we store in $\boldsymbol{W}$.

PSDb Subsystem $\boldsymbol{s}_b$ is evaluated first, yielding $\boldsymbol{W}_{ba} = \boldsymbol{0}$.

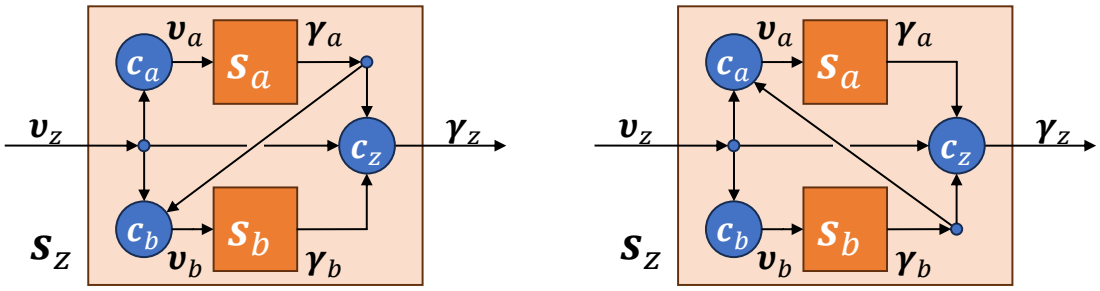
For the remaining submatrices in $\boldsymbol{W}$ we do not impose any restrictions. The cases *PSDa* and *PSDb* lead to the connection matrices

$$\boldsymbol{W}_{PSDa} = \begin{array}{c} \\ \boldsymbol{c}_a \\ \boldsymbol{c}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccc} \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_b & \boldsymbol{v}_z \end{array} \\ \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{W}_{az} \\ \boldsymbol{W}_{ba} & \boldsymbol{0} & \boldsymbol{W}_{bz} \\ \boldsymbol{W}_{za} & \boldsymbol{W}_{zb} & \boldsymbol{W}_{zz} \end{bmatrix} \end{array} \tag{3.26}$$

and

$$\boldsymbol{W}_{PSDb} = \begin{array}{c} \\ \boldsymbol{c}_a \\ \boldsymbol{c}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccc} \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_b & \boldsymbol{v}_z \end{array} \\ \begin{bmatrix} \boldsymbol{0} & \boldsymbol{W}_{ab} & \boldsymbol{W}_{az} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{W}_{bz} \\ \boldsymbol{W}_{za} & \boldsymbol{W}_{zb} & \boldsymbol{W}_{zz} \end{bmatrix} \end{array}. \tag{3.27}$$

The resulting combined models can further be visualized for a better understanding, see Fig. 3.17.



(a) The *PSDa* combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The system $\boldsymbol{s}_a$ is evaluated first, then the system $\boldsymbol{s}_b$ is evaluated based on the intermediate results of $\boldsymbol{s}_a$.

(b) The *PSDb* combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The system $\boldsymbol{s}_b$ is evaluated first, then the system $\boldsymbol{s}_a$ is evaluated based on the intermediate results of $\boldsymbol{s}_b$.

Figure 3.17: The two different cases for the *PSD* architecture. Both subsystems access the combined model input $\boldsymbol{v}_z$. The final system output depends on results from $\boldsymbol{s}_a$, $\boldsymbol{s}_b$ and the system input $\boldsymbol{v}_z$ (via DFT).

To check the solvability of the system, the dependency matrices can be investigated. As to expect, the system dependency matrix for case *PSDa* can be transformed to LT form:

$$\boldsymbol{D}_{PSDa} = \begin{array}{c} \\ \boldsymbol{s}_a \\ \boldsymbol{s}_b \\ \boldsymbol{c}_a \\ \boldsymbol{c}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_b & \boldsymbol{v}_a & \boldsymbol{v}_b & \boldsymbol{y} \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{D}_a & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{D}_b & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}_{ba} & \boldsymbol{0} & \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} \\ \boldsymbol{D}_{za} & \boldsymbol{D}_{zb} & \boldsymbol{0} & \boldsymbol{0} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array} \overset{\text{LT}}{=} \begin{array}{c} \\ \boldsymbol{c}_a \\ \boldsymbol{s}_a \\ \boldsymbol{c}_b \\ \boldsymbol{s}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{v}_a & \boldsymbol{\gamma}_a & \boldsymbol{v}_b & \boldsymbol{\gamma}_b & \boldsymbol{y} \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}_a & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_{ba} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{D}_b & \underline{\boldsymbol{I}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_{za} & \boldsymbol{0} & \boldsymbol{D}_{zb} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array}. \tag{3.28}$$

The success of the LT transformation corresponds to finding an order for solving the resulting system of equations from top (first equation in the system) to bottom

(last equation), one equation after another. The same is possible for the case *PSDb*:

$$\boldsymbol{D}_{PSDb} = \begin{array}{c} \\ \boldsymbol{s}_a \\ \boldsymbol{s}_b \\ \boldsymbol{c}_a \\ \boldsymbol{c}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_b & \boldsymbol{v}_a & \boldsymbol{v}_b & \boldsymbol{\gamma}_z \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{D}_a & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{D}_b & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_{ab} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \underline{\boldsymbol{I}} & \boldsymbol{0} \\ \boldsymbol{D}_{za} & \boldsymbol{D}_{zb} & \boldsymbol{0} & \boldsymbol{0} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array} \overset{\mathsf{LT}}{=} \begin{array}{c} \\ \boldsymbol{c}_b \\ \boldsymbol{s}_b \\ \boldsymbol{c}_a \\ \boldsymbol{s}_a \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{v}_b & \boldsymbol{\gamma}_b & \boldsymbol{v}_a & \boldsymbol{\gamma}_a & \boldsymbol{\gamma}_z \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}_b & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_{ab} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{D}_a & \underline{\boldsymbol{I}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{D}_{zb} & \boldsymbol{0} & \boldsymbol{D}_{za} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array}. \tag{3.29}$$

This results in a system free of algebraic loops for both cases, independent of the chosen connections in $\boldsymbol{W}_{ba}$ (or $\boldsymbol{W}_{ab}$), $\boldsymbol{W}_{za}$ and $\boldsymbol{W}_{zb}$ as well as of the dependency matrices of the original subsystems $\boldsymbol{D}_a$ and $\boldsymbol{D}_b$.

Contribution 2 *Under the assumption that the dependencies inside two algebraic systems are unknown (so the worst case is assumed), the* PSD *architecture is derived, that is the most flexible way of combining two systems of algebraic equations, so that the resulting system is guaranteed to be free of algebraic loops induced by the connection equations. This results in the opportunity to parameterize and/or optimize for connections between two HUDA-ODEs without the need to check for, localize, and solve additional algebraic loops.*

3.4.4 Discontinuities

As introduced, besides algebraic loops, one major challenge are discontinuities. If an event happens, a new *local state* is determined inside of one of the subsystems. This is a natural consequence of modeling: Events are defined within the scope of the subsystem, and do not involve further assumptions about what happens outside the borders of the subsystem. For example in case of a physical subsystem, the event definition has some explainable meaning and is modeled on purpose (e.g. ball hits the ground), but this definition is not valid for a combined model anymore (e.g. an additional ML model could shift the ground). Investigating the system of equations, during event handling a part of the causalization changes: In general, the combined system state $\boldsymbol{x}_z$ is obtained by numerical integration, is therefore *known* and not part of the dependency matrix columns. However, at the moment a discontinuity happens, one of the subsystems determines a new local state (that becomes now known) and the global system state $\boldsymbol{x}_z$ becomes unknown. This new local state needs to be propagated through the entire combined system back to the ODE solver to proceed with the integration of a global system state.

This problem can further be examined by discussing the system dependency matrix in case of a state event in one of the subsystems. In the following, only the architecture *PSDa* is investigated, but a strategy for *PSDb* can be derived in analogy. Two cases for *PSDa* can be distinguished:

(a) An event is triggered in subsystem $\boldsymbol{s}_a$, leading to a new local state $\boldsymbol{x}_a$. Because for the architecture *PSDa*, subsystem $\boldsymbol{s}_a$ is evaluated before $\boldsymbol{s}_b$, this case is easier to handle because it does not require the propagation of the new local state $\boldsymbol{x}_a$ through the subsystem $\boldsymbol{s}_b$ to obtain $\boldsymbol{x}_z$.

(b) An event is triggered in subsystem $\boldsymbol{s}_b$, leading to a new local state $\boldsymbol{x}_b$. Because for the architecture *PSDa*, subsystem $\boldsymbol{s}_a$ is evaluated before $\boldsymbol{s}_b$, this case is

more complex to handle because it is required to propagate the new local state $\boldsymbol{x}_b$ through the subsystem $\boldsymbol{s}_a$ to obtain $\boldsymbol{x}_z$.

For sake of readability, but without limitation of general validity, we further simplify that $\boldsymbol{v}_\phi = \boldsymbol{x}_\phi$ for $\phi \in \{a, b, z\}$, resulting in systems whose only known/input argument is the corresponding system state. The resulting *PSDa* architecture before event handling is illustrated in Fig. 3.18.

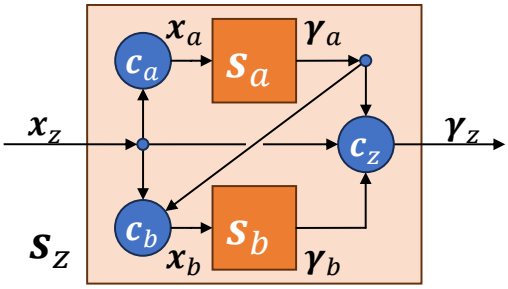


Figure 3.18: *PSDa* without an event happening: The state $\boldsymbol{x}_z$ is separated (not necessarily disjointly) into $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$. Both subsystems calculate outputs $\boldsymbol{\gamma}_a$ and $\boldsymbol{\gamma}_b$ that are finally combined to $\boldsymbol{\gamma}_z$.

Events in subsystem $\boldsymbol{s}_a$

In case of an event for the subsystem $\boldsymbol{s}_a$, $\boldsymbol{x}_z$ (formally determined by the numerical integrator) becomes unknown and $\boldsymbol{x}_a$ is determined by the event affect function of subsystem $\boldsymbol{s}_a$. The changing causalization is visualized in Fig. 3.19.

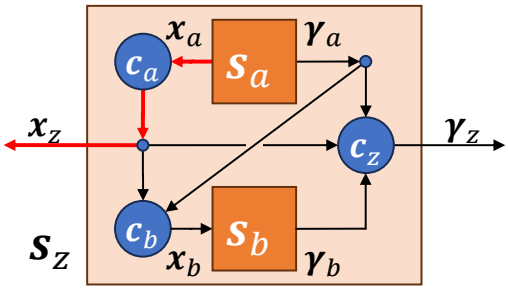


Figure 3.19: *PSDa* with event in subsystem $\boldsymbol{s}_a$, changes in causalization are visualized in red. Now, the subsystem $\boldsymbol{s}_a$ computes a new local state $\boldsymbol{x}_a$, that must be propagated through $\boldsymbol{c}_a$ to obtain the new global state $\boldsymbol{x}_z$.

The resulting dependency matrix for *PSDa* with event in $\boldsymbol{s}_a$ therefore states:

$$\boldsymbol{D}^{(a)}_{PSDa} \overset{\text{BLT}}{=} \begin{array}{c} \\ \boldsymbol{s}_a \\ \boldsymbol{c}_a \\ \boldsymbol{c}_b \\ \boldsymbol{s}_b \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \boldsymbol{\gamma}_a & \boldsymbol{x}_z & \boldsymbol{x}_b & \boldsymbol{\gamma}_b & \boldsymbol{\gamma}_z \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \underline{\boldsymbol{D}_{az}} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}_{ba} & \boldsymbol{D}_{bz} & \underline{\boldsymbol{I}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{D}_b & \underline{\boldsymbol{I}} & \boldsymbol{0} \\ \boldsymbol{D}_{za} & \boldsymbol{D}_{zz} & \boldsymbol{0} & \boldsymbol{D}_{zb} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array}. \tag{3.30}$$

Note, that the system is not in LT-shape — because $\boldsymbol{D}_{az}$ is not required to be in LT shape — and is therefore featuring an algebraic loop (orange).

Note, that this requires solving a linear system of equations for $\boldsymbol{x}_z$, because it is not guaranteed that $\boldsymbol{c}_a$ — which boils down to the submatrix $\boldsymbol{W}_{az}$ — is invertible:

$$\mathbf{0} = \boldsymbol{c}_a(\boldsymbol{x}_z) - \boldsymbol{x}_a = \boldsymbol{W}_{az}\boldsymbol{x}_z + \boldsymbol{b}_a - \boldsymbol{x}_a. \tag{3.31}$$

However, if $\boldsymbol{W}_{az}$ is invertible, one can find the solution analytically, of course:

$$\boldsymbol{x}_z = \boldsymbol{W}_{az}^{-1}(\boldsymbol{x}_a - \boldsymbol{b}_a). \tag{3.32}$$

Events in subsystem $\boldsymbol{s}_b$
As already stated, in case of an event for the subsystem $\boldsymbol{s}_b$, $\boldsymbol{x}_z$ (formally determined by the numerical integrator) becomes unknown and $\boldsymbol{x}_b$ is determined by the event affect function of subsystem $\boldsymbol{s}_b$. Again, we can visualize the change in the causalization as in Fig. 3.20.

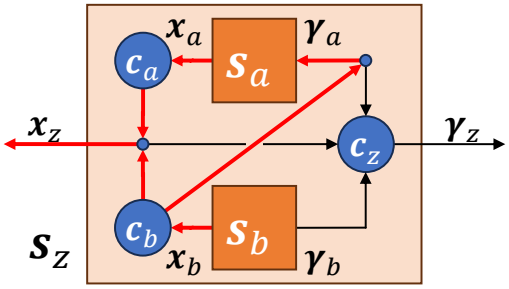


Figure 3.20: *PSDa* with event in subsystem $\boldsymbol{s}_b$, changes in causalization are visualized in red. Now, the subsystem $\boldsymbol{s}_b$ computes a new local state $\boldsymbol{x}_b$, that must be propagated through two different traces to obtain the new global state $\boldsymbol{x}_z$: First, via $\boldsymbol{c}_b$ to obtain $\boldsymbol{x}_z$ and second via $\boldsymbol{c}_b$, $\boldsymbol{s}_a$ and $\boldsymbol{c}_a$.

The dependency matrix in this case states:

$$\boldsymbol{D}^{(b)}_{PSDa} \overset{\text{BLT}}{=} \begin{array}{c} \\ \boldsymbol{s}_b \\ \boldsymbol{c}_b \\ \boldsymbol{s}_a \\ \boldsymbol{c}_a \\ \boldsymbol{c}_z \end{array} \begin{array}{c} \begin{array}{ccccc} \gamma_b & \gamma_a & \boldsymbol{x}_a & \boldsymbol{x}_z & \gamma_z \end{array} \\ \begin{bmatrix} \underline{\boldsymbol{I}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \underline{\boldsymbol{D}_{ba}} & \mathbf{0} & \boldsymbol{D}_{bz} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} & \underline{\boldsymbol{D}_a} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \underline{\boldsymbol{D}_{az}} & \mathbf{0} \\ \boldsymbol{D}_{zb} & \boldsymbol{D}_{za} & \mathbf{0} & \boldsymbol{D}_{zz} & \underline{\boldsymbol{I}} \end{bmatrix} \end{array}. \tag{3.33}$$

Note, that the system is not in LT-shape and is featuring a (large) algebraic loop (orange). This requires solving a nonlinear system of equations for $\boldsymbol{x}_z$, in case of a nonlinear subsystem $\boldsymbol{s}_a$:

$$\begin{aligned} \mathbf{0} &= \boldsymbol{c}_b(\boldsymbol{s}_a(\boldsymbol{c}_a(\boldsymbol{x}_z)), \boldsymbol{x}_z) - \boldsymbol{x}_b && (3.34)\\ &= \boldsymbol{W}_{ba}\boldsymbol{s}_a(\boldsymbol{c}_a(\boldsymbol{x}_z)) + \boldsymbol{W}_{bz}\boldsymbol{x}_z + \boldsymbol{b}_b - \boldsymbol{x}_b && (3.35)\\ &= \underbrace{\boldsymbol{W}_{ba}\boldsymbol{s}_a(\boldsymbol{W}_{az}\boldsymbol{x}_z + \boldsymbol{b}_a)}_{\text{via } \boldsymbol{c}_b,\ \boldsymbol{s}_a \text{ and } \boldsymbol{c}_a} + \underbrace{\boldsymbol{W}_{bz}\boldsymbol{x}_z}_{\text{via } \boldsymbol{c}_b} + \boldsymbol{b}_b - \boldsymbol{x}_b. && (3.36) \end{aligned}$$

Note, that for more specific cases, like parts of the weight matrix are zero, analytical solutions can be derived. For sparse connection matrices, the problem should

be in good condition, meaning only the dependent part of the state vector is modified, to prevent unintended state changes in the non-dependent part of the state. To ensure this, local SA could be applied beforehand, to determine the sensitive entries of $\boldsymbol{x}_b$ with respect to $\boldsymbol{x}_z$. The problem needs to be reformulated so that the nonlinear solver (or optimizer[12]) is only allowed to change the sensitive entries of $\boldsymbol{x}_z$.

Contribution 3 *We further extended the* PSD *architecture for HUDA-ODEs by a generic strategy to deal with the changing causalization during event handling. This measure takes hybrid modeling involving event systems one step further toward the industrial application.*

In the following we show, that more simple architectures — including the known concepts of serial and parallel connections — can be further derived from the *PSD* architecture by permutation of the following three attributes: Parallel (P), Sequential (S) and DFT (D).

3.4.5 Derived architectures

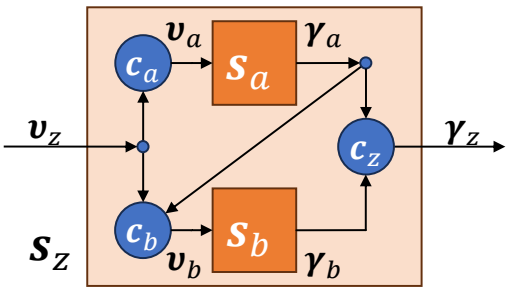
Based on the *PSD* architecture, different architectures can be derived by setting further parts of the connection matrix $\boldsymbol{W}$ to zero, compare Tab. 3.2. Note, that this is equivalent to removing connections from the *PSD* architecture. Since all architectures generated in this way feature only a subset of the connections of *PSD*, all derived architectures also have no algebraic loops by connections and are able to handle discontinuous subsystems. However, further simplifications to the proposed strategies can be applied for specific sub-architectures. A subset of these architectures (*PS*, *P* and *S*) is briefly introduced in the following.

Table 3.2: Occupancy of connection matrices for the different derived architectures. Legend: $\times$ = true, $\circ$ = false, $(x, y) = (a, b)$ for case a ($\boldsymbol{s}_a$ before $\boldsymbol{s}_b$) and $(x, y) = (b, a)$ for case b ($\boldsymbol{s}_b$ before $\boldsymbol{s}_a$). $\boldsymbol{W}_{xy} = \boldsymbol{0}$, $\boldsymbol{W}_{xx} = \boldsymbol{0}$ and $\boldsymbol{W}_{yy} = \boldsymbol{0}$ for all architectures.

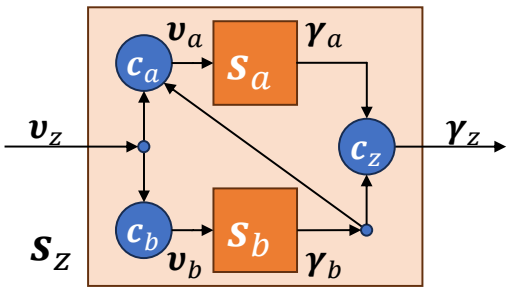
P	S	D	$\boldsymbol{W}_{xz} \neq \boldsymbol{0}$	$\boldsymbol{W}_{yx} \neq \boldsymbol{0}$	$\boldsymbol{W}_{yz} \neq \boldsymbol{0}$	$\boldsymbol{W}_{zx} \neq \boldsymbol{0}$	$\boldsymbol{W}_{zy} \neq \boldsymbol{0}$	$\boldsymbol{W}_{zz} \neq \boldsymbol{0}$
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
$\times$	$\times$	$\circ$	$\times$	$\times$	$\times$	$\times$	$\times$	$\circ$
$\times$	$\circ$	$\times$	$\times$	$\circ$	$\times$	$\times$	$\times$	$\times$
$\times$	$\circ$	$\circ$	$\times$	$\circ$	$\times$	$\times$	$\times$	$\circ$
$\circ$	$\times$	$\times$	$\times$	$\times$	$\circ$	$\circ$	$\times$	$\times$
$\circ$	$\times$	$\circ$	$\times$	$\times$	$\circ$	$\circ$	$\times$	$\circ$
$\circ$	$\circ$	$\times$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\times$
$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$

[12]The problem is balanced regarding the number of knowns and unknowns, allowing for the application of a nonlinear solver. However, the zero finding problem can also be reformulated as optimization problem (e.g. by squaring the zero-crossing residual) and be solved via optimization.

PS: Parallel-sequential
The *PS* architecture evaluates the first subsystem based on the input of the combined system, and then the second subsystem based on the intermediate result of the first system, as well as the original input of the combined system. This way the *PS* architecture features characteristics of both the parallel (both subsystems are connected to the combined system input) as well as the sequential (one subsystem is evaluated before the other) architecture. Comparing to the *PSD* architecture, no DFT is present.



(a) The *PSa* combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The system $\boldsymbol{s}_a$ is evaluated first, then the system $\boldsymbol{s}_b$ is evaluated based on the intermediate results of $\boldsymbol{s}_a$ and the combined system input $\boldsymbol{v}_z$.



(b) The *PSb* combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The system $\boldsymbol{s}_b$ is evaluated first, then the system $\boldsymbol{s}_a$ is evaluated based on the intermediate results of $\boldsymbol{s}_b$ and the combined system input $\boldsymbol{v}_z$.

Figure 3.21: The two cases for the *PS* architecture. Both subsystems access the combined model input $\boldsymbol{v}_z$. The final combined system output depends on both results from $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$.

P: Parallel
The parallel architecture evaluates two subsystems side by side. Because they are not sharing intermediate results with each other, it is not necessary to decide which of the subsystems is evaluated first.

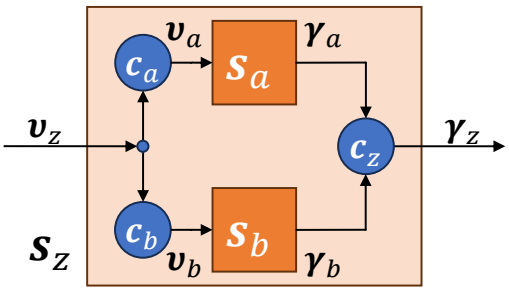
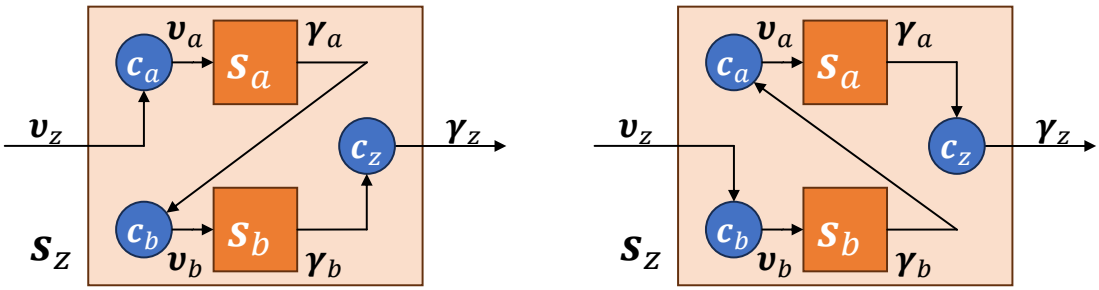


Figure 3.22: The *P*(arallel) combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The input $\boldsymbol{v}_z$ is separated (not necessarily uniquely) into $\boldsymbol{v}_a$ and $\boldsymbol{v}_b$. Both subsystems calculate outputs $\boldsymbol{\gamma}_a$ and $\boldsymbol{\gamma}_b$ that are finally combined to $\boldsymbol{\gamma}_z$.

It is worth mentioning, that because of $\boldsymbol{W}_{ba} = \boldsymbol{0}$ for the architecture *P*, the nonlinear system of equations to be solved for events in $\boldsymbol{s}_b$ (see Eq. 3.34) simplifies to a linear system, just like as for an event in $\boldsymbol{s}_a$ (see Eq. 3.31):

$$\boldsymbol{0} = \boldsymbol{c}_b(\boldsymbol{x}_z) - \boldsymbol{x}_b = \boldsymbol{W}_{bz}\boldsymbol{x}_z + \boldsymbol{b}_b - \boldsymbol{x}_b. \tag{3.37}$$

S: Sequential / Serial
The sequential architecture strictly evaluates one subsystem after the other. The subsystem that is evaluated first passes its output to the second subsystem.



(a) The *Sa* combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The system $\boldsymbol{s}_a$ is evaluated first, then the system $\boldsymbol{s}_b$ is evaluated based on the intermediate results of $\boldsymbol{s}_a$. The final system output depends only on results from $\boldsymbol{s}_b$.

(b) The *Sb* combination of two algebraic systems $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$. The system $\boldsymbol{s}_b$ is evaluated first, then the system $\boldsymbol{s}_a$ is evaluated based on the intermediate results of $\boldsymbol{s}_b$. The final system output depends only on results from $\boldsymbol{s}_a$.

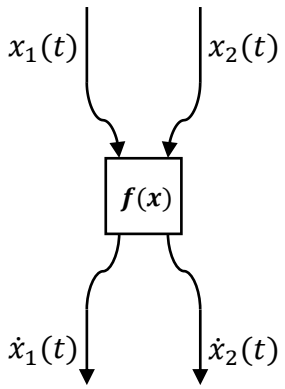
Figure 3.23: The two cases for the *S* architecture.

Contribution 4 *Besides the generic* PSD *architecture, we showed that the well-known parallel (*P*) and serial (*S*) architectures can be fully expressed within the derived combination method, besides other new options (like e.g.* PS*).*

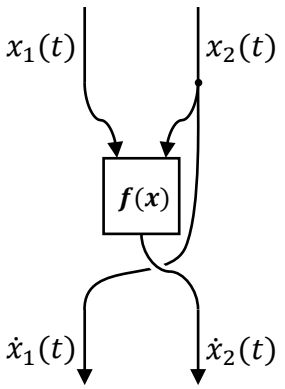
3.4.6 Second order preservation (SOP)

In various domains (especially in mechanics), double-differentiating systems are common, i.e. systems that differentiate at least one state two times. In translational mechanics, this corresponds to the relationship between position and velocity (first differentiation), and velocity and acceleration (second differentiation). In practical terms, this means that the first derivative of the quantity (in its role as state derivative) is not calculated as part of the right-hand side, but forwarded directly (in its role as state), compare Fig. 3.24.

Since the core idea behind SciML is to make meaningful use of prior knowledge, it seems worthwhile to embed prior knowledge about the order of differentiation into the hybrid model as well. This idea is not novel, e.g. modifying the concept of neural ODEs (or augmented neural ODEs) to efficiently learn for second-order ODEs is discussed in Norcliffe *et al.* [88]. This concept can also be applied for hybrid modeling, or HUDA-ODEs in particular. If we compare the first- and second-order ODEs (again, see Fig. 3.24), we find that the core difference is that only a subset of state derivatives is determined by the right-hand side $\boldsymbol{f}(\boldsymbol{x})$, whereas the remaining derivatives are passed through for the second-order ODE. This *connection pattern* can easily be expressed by the introduced connection matrices: States directly connected to derivatives result in occupying the corresponding entries of the connection matrix $\boldsymbol{W}$, and the original right-hand side is used to only calculate the two-times differentiated variables. Using the translational mechanical system as example again, the right-hand side only calculates the *acceleration*, the *velocity* is



(a) A common (first-order) ODE with two states. The state derivatives $\dot{x}_1$ and $\dot{x}_2$ are calculated based on the states x_1 and x_2.



(b) A second-order ODE with two states. Only the state derivative $\dot{x}_2$ is calculated based on the states x_1 and x_2, the state derivative $\dot{x}_1$ equals the state x_2 and is forwarded directly.

Figure 3.24: Example for a first- and second-order ODE.

directly forwarded to the ODE solver and both are integrated to the velocity and position for the next time instant. We note that we can apply this in principle for all architectures presented (so *PSD* and its derivatives), and mark architectures with a second-order preserving structure with the abbreviation "SOP". Even if such ODEs are less common in engineering, this approach can be extended to higher orders of differentiation, too, of course.

Contribution 5 *We introduced the concept of second order preservation (SOP) for hybrid models, that not only enhances the physical correctness by preserving the integrative relationship between physical variables, but also slightly reduces the amount of parameters by reducing the size of the output layer (only half the outputs are required for a second-order model). Both aspects further enhance training stability and generalizability.*

3.5 Experiment

In the following, the proposed method for hybrid modeling, including handling discontinuities, is validated at a hybrid model based on the bouncing ball (compare Sec. 2.4). In this experiment, we compare different connection patterns for hybrid model combination. We start by making some preliminary considerations on which architectures should be able to solve the proposed problem, perform the experiment and finally validate the results.

The overall goal of this experiment is to learn for a not modeled physical effect, nonlinear air friction. We build the hybrid model based on the physical simulation model of the bouncing ball (see Sec. 2.4), but remove the nonlinear air friction term. We extend the bouncing ball without modeled air friction by a simple FFNN, which in turn is supposed to learn the air friction effect. The combination of physical and ML model is performed by applying the proposed architectures one after another. We compare training results of hybrid models, that where trained on a single, two, three, or four entirely different trajectories of the bouncing ball, which can be in-

vestigated in Sec. A.3. In summary, we change the connection architectures and the number of training trajectories — which corresponds to the *amount of data* — across the individual experiments. This is to examine correlations between the chosen architecture and the ability to generalize.

The hybrid model, as well as the physical and ML submodels, are expressed as HUDA-ODEs. The hybrid model inherits all equations (event indicator, event affect, output) except the ODE $\boldsymbol{f}$ from the physical model. The function $\boldsymbol{f}$ is obtained by model combination of the physical submodel function $\boldsymbol{f}$ (the right-hand side of the bouncing ball) and the ML submodel function $\boldsymbol{g}$ (inference of the FFNN). Further, we apply a SOP pattern for all architectures, meaning the FFNN does only compute the ball acceleration in both directions, instead of the entire state derivative which further includes the velocities. The velocities are fed through directly to preserve the second order ODE in a structural way. The training setup is summarized in Tab. 3.3.

Table 3.3: Setup for the architecture comparison experiment.

Attribute	Value
Training strategy	Growing Horizon (SB)
Horizon start	0.05 s
Horizon loss threshold	0.06
Horizon increment	0.01 s
Horizon length	2.1 s
Loss function	MAE (s_x, v_x, s_y, v_y)
Loss evaluation points	$\{0.0\,\text{s}, 0.01\,\text{s}, ..., 2.09\,\text{s}, 2.1\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam w. exp. decay
	($\eta_{start} = 0.01$, $\eta_{stop} = 0.001$, $\lambda = 0.0005$)
Architecture	*PSDa*, *PSa*, *PD*, *P*, *SDa*, *Sa* and *D* (SOP)
ANN layout	$4 \rightarrow 8$ (tanh)
	$8 \rightarrow 2$ (tanh)
Number of training steps	10,000
Number of repetitions	100

This experiment is performed similarly, but with significantly fewer details, within Thummerer and Mikelsons [84]. In the following, a hypothesis is set up before investigation of the experimental results.

3.5.1 Hypothesis

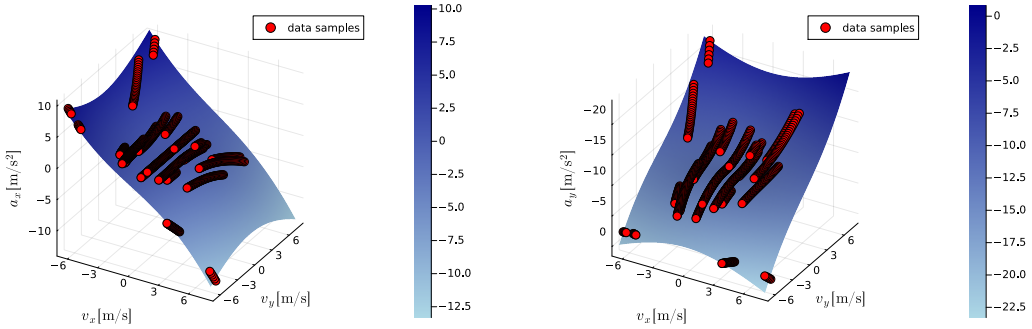
In this experiment, the following architectures are compared: *PSDa*, *PSa*, *PD*, *P*, *SDa*, *Sa* and *D*. By investigating the original right-hand side of the bouncing ball, we find that the force by air friction

$$f_{\phi}^{(air)} = -|\boldsymbol{v}| \cdot v_{\phi} \cdot \mu \quad \text{for} \quad \phi \in \{x, y\} \tag{3.38}$$

can be divided by the mass m (which is a constant parameter) to obtain the corresponding fraction of acceleration caused by air friction and be *added* to the ball

acceleration in directions x and y. This directly implies, that architectures including *parallel* layouts, that can express additions between multiple models, should be capable of learning for the addressed effect.

Further, if we plot the velocities of the physical model against the acceleration (Fig. 3.25), we find that the acceleration in x as well as y can be fully determined by both the velocities v_x and v_y. This directly results in the insight, that also *serial*



(a) Accelerations in x direction.

(b) Accelerations in y direction.

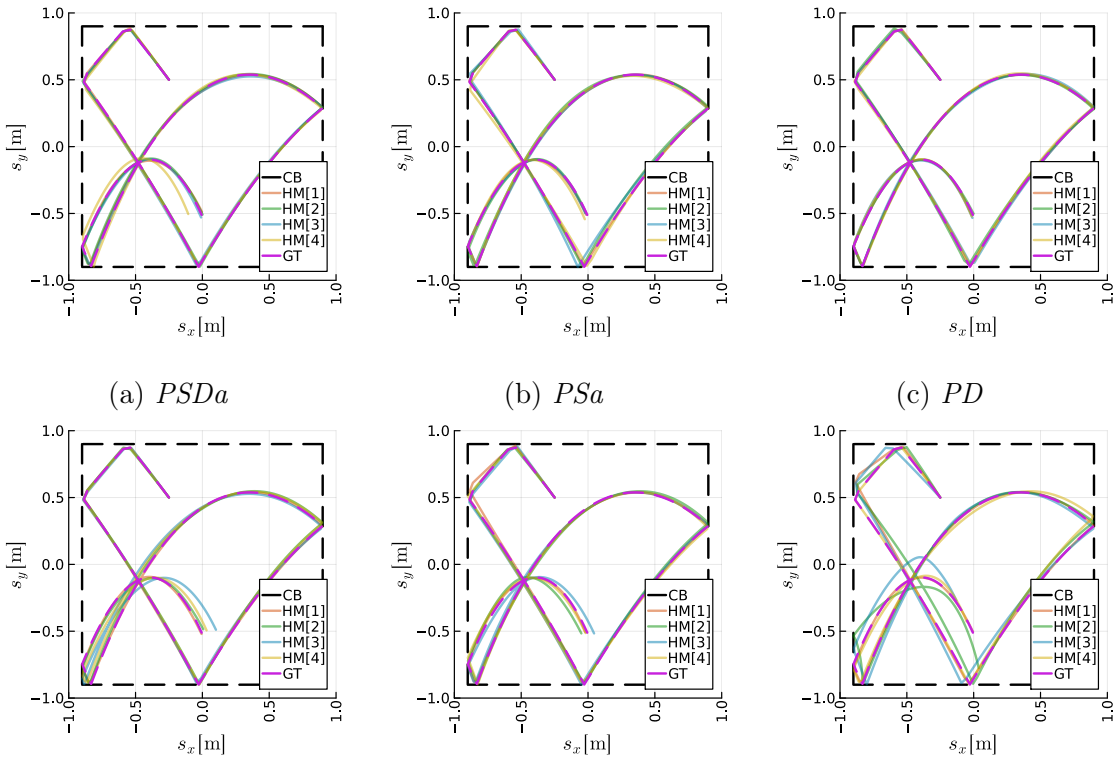
Figure 3.25: Correlations between the velocities and the accelerations in x and y in data of the bouncing ball including air friction. Note that the function evaluation leads to a smooth and closed surface and that every pair of v_x and v_y uniquely maps on an acceleration value. The data samples used for training are given in red.

architectures should be capable of learning the desired friction equation, because the correct acceleration can be derived directly from the velocity (state derivative). Whereas this relation is a trivial insight, because we can validate this by investigating Eq. 3.38, in general the effect being learned is not known and can therefore not be investigated on a symbolic level. The corresponding plot for a preliminary examination, however, can be produced solely based on data.

Finally, we conclude that DFT alone is not able to express the nonlinear friction effect, because the bouncing ball acceleration can not be fully expressed by a linear combination of the system state (position and velocity). To conclude our observations, we expect all architectures — except D — to learn for a proper hybrid model.

3.5.2 Results & discussion

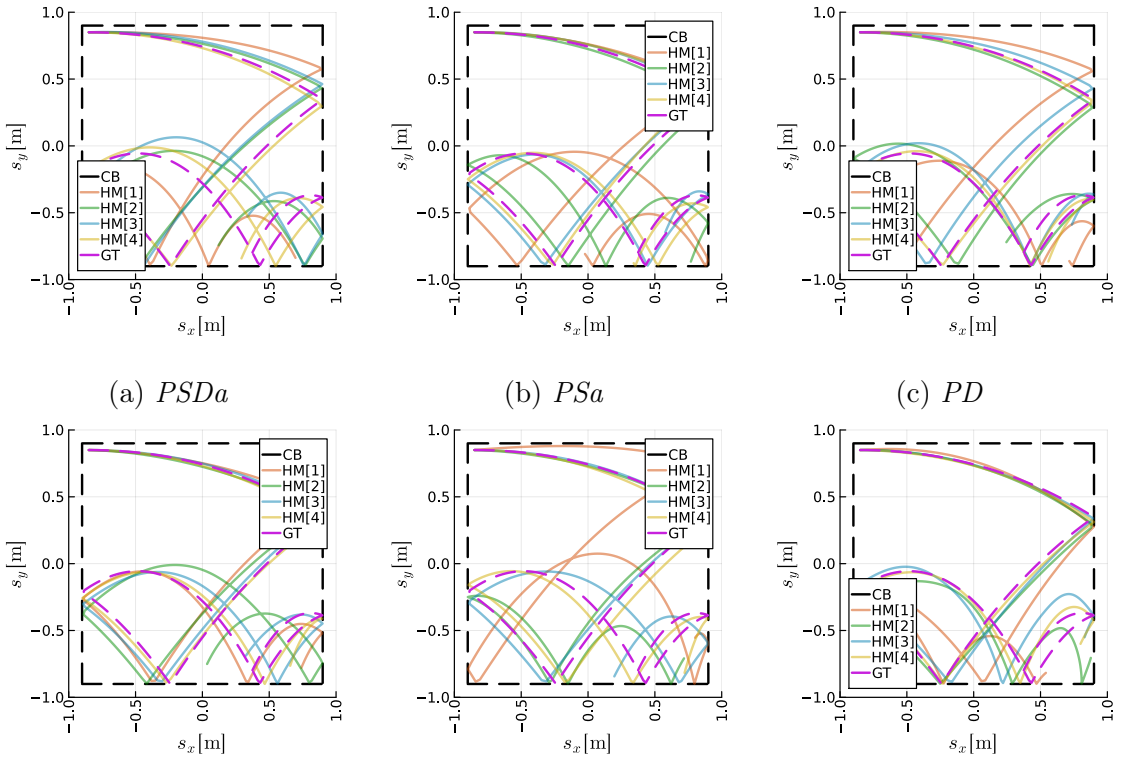
After training the architectures under the introduced conditions, the results in Fig. 3.26 are obtained. The validity of the hypothesis can be confirmed by investigation of the plots. All architectures involving the parallel or serial patterns are able to learn for the nonlinear friction model as expected. Solely the architecture D can not be trained into a solvable system — which matches our expectation regarding the non-existence of an acceptable solution for this case. Qualitative results for the architecture D are therefore omitted, but are part of the upcoming quantitative examinations. The successful training motivates to further check (unknown) test data, compare Fig. 3.27.



(a) *PSDa* (b) *PSa* (c) *PD*

(d) *P* (e) *SDa* (f) *Sa*

Figure 3.26: Comparison of the closest-to-median simulation trajectories of different architectures on training data. For the hybrid models (HMs), a different amount of data (number of trajectories) is used for training (square brackets in the legend). Further, the collision border (CB) is shown and trajectories for the ground truth (GT) are given. As to expect, all shown architectures are able to qualitatively learn for the ground truth data.



(a) *PSDa* (b) *PSa* (c) *PD*

(d) *P* (e) *SDa* (f) *Sa*

Figure 3.27: Comparison of the closest-to-median simulation trajectories of different architectures on test data. For the hybrid models (HMs), a different amount of data (number of trajectories) is used for training (square brackets in the legend). Further, the collision border (CB) is shown and trajectories for the ground truth (GT) are given.

Even if the results on training data are similar, we find stronger differences between the architectures on testing data. Because the precision on testing data in general increases if more trajectories where used during training, we conclude that bad extrapolation directly results from the lack of data, which is a common observation in ML. The amount of data that is used for training is not enough to exclude false correlations beyond doubt. Quantities of the physical system correlate, even if there is no causal dependency, but the machine learning model itself is not able to distinguish between correlation and causality.

However, if we restrict the architecture, which is nothing but incorporation of knowledge, we also prevent learning of some of these non-causal correlations. The overall trend on testing data is, that less expressive architectures extrapolate better, the median error is for example larger in *PSDa* than in *PSa*, *PD* and *P*, independent of the number of used training trajectories, see Fig. 3.28.

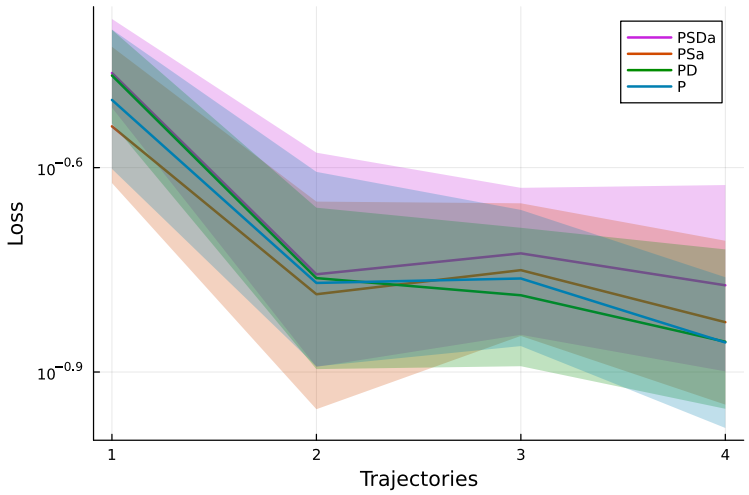


Figure 3.28: Median test loss depending on the number of training trajectories for the architecture experiment. The semi-transparent ribbons give the corresponding 25th and 75th percentile. Incorporating more training trajectories (horizontal axis) leads to a better loss value on testing data (vertical axis). Less expressive architectures feature smaller loss values.

We can assume that keeping the architecture more flexible is a direct tradeoff for extrapolation capability — if only very little data is available. In theory, this deviation between the architectures should vanish, if a *sufficiently* large dataset is available and can be trained on *sufficiently* long, but this experiment nicely shows the positive and negative impact of the chosen architectures subject to limited data and limited training resources.

Investigating Tab. 3.4, we find slightly increased loss values for training along all architectures, if we increase the number of training trajectories. This is in line with our intuition, because the same model needs to fit multiple trajectories and not only a single one that could be overfitted.

Table 3.4: Quantitative results of the bouncing ball hybrid model applying different architectures and varying the amount of data (trajectories). The architecture *D* leads to a non-solvable (n.s.) hybrid model.

Arch.	Traj.	Median (train)	P_{25} | P_{75} (train)	Median (test)	P_{25} | P_{75} (test)
PSDa	1	0.018	[0.013, 0.021]	0.346	[0.308, 0.415]
PSDa	2	0.024	[0.019, 0.031]	0.175	[0.128, 0.264]
PSDa	3	0.034	[0.029, 0.042]	0.188	[0.143, 0.235]
PSDa	4	0.047	[0.04, 0.057]	0.169	[0.126, 0.237]
PSa	1	0.019	[0.016, 0.024]	0.289	[0.238, 0.378]
PSa	2	0.029	[0.023, 0.039]	0.164	[0.111, 0.224]
PSa	3	0.034	[0.028, 0.045]	0.178	[0.142, 0.223]
PSa	4	0.047	[0.037, 0.061]	0.149	[0.113, 0.196]
PD	1	0.016	[0.012, 0.02]	0.343	[0.292, 0.4]
PD	2	0.023	[0.019, 0.033]	0.173	[0.127, 0.219]
PD	3	0.031	[0.026, 0.037]	0.163	[0.128, 0.205]
PD	4	0.039	[0.032, 0.047]	0.139	[0.111, 0.191]
P	1	0.015	[0.012, 0.02]	0.316	[0.25, 0.401]
P	2	0.026	[0.022, 0.036]	0.17	[0.128, 0.248]
P	3	0.034	[0.028, 0.044]	0.173	[0.137, 0.218]
P	4	0.038	[0.032, 0.044]	0.139	[0.104, 0.173]
SDa	1	0.027	[0.02, 1.399]	0.509	[0.392, 1.133]
SDa	2	0.033	[0.023, 0.037]	0.183	[0.137, 0.285]
SDa	3	0.034	[0.028, 0.044]	0.187	[0.15, 0.227]
SDa	4	0.051	[0.043, 0.065]	0.161	[0.121, 0.23]
Sa	1	0.04	[0.031, 0.861]	0.344	[0.183, 0.448]
Sa	2	0.095	[0.057, 0.276]	0.235	[0.163, 0.33]
Sa	3	0.082	[0.057, 0.309]	0.223	[0.148, 0.354]
Sa	4	0.071	[0.053, 0.093]	0.196	[0.136, 0.245]
D	1	n.s.	n.s.	n.s.	n.s.
D	2	n.s.	n.s.	n.s.	n.s.
D	3	n.s.	n.s.	n.s.	n.s.
D	4	n.s.	n.s.	n.s.	n.s.

To conclude, keeping the structure of the hybrid model fully trainable offers a valuable option if only very little is known about the system, but requires (even for academic examples) significantly more training data to extract the causal effect and neglect non-causal correlations. By providing more information regarding the structure, the hybrid model is able to learn for the actual causality and extrapolates better.

Interpretability

Finally, we want to investigate the interpretability of the combined model. Because some architectures (like P) extrapolate better than others (like $PSDa$), we assume to find traces for this in the visualization of the connection matrices, see Fig. 3.29-3.36. In the following, we only investigate a single run of the experiment — the one closest to the median loss — because there exist multiple acceptable minima and plotting mean values would therefore result in noisy matrices. For a given submatrix $\boldsymbol{W}_{xy}$ and a scalar matrix entry $\boldsymbol{W}_{ij} \in \boldsymbol{W}_{xy}$, we use a linear color mapping ranging from $W_{ij} = 0$ (black) to $\|W_{ij}\| = max(\|\boldsymbol{W}\|)$ (white), but scale matrices only if they contain values > 1. We give the applied scaling factor in the plot title. In plain language, this results in non-existent connections being visualized in black, whereas *relative strong* connections are visualized in white, and *relative weak* connections in between (gray).

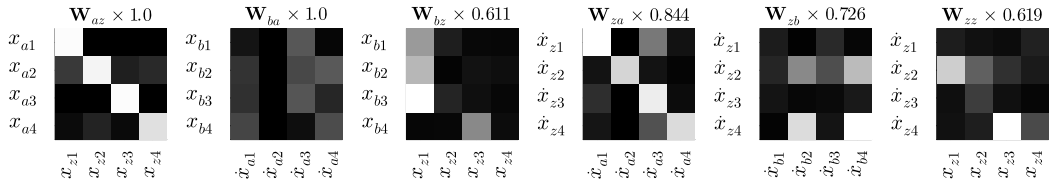


Figure 3.29: *PSDa* (1 training trajectory)

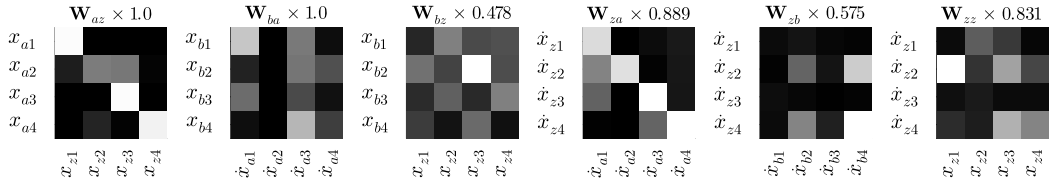


Figure 3.30: *PSDa* (2 training trajectories)

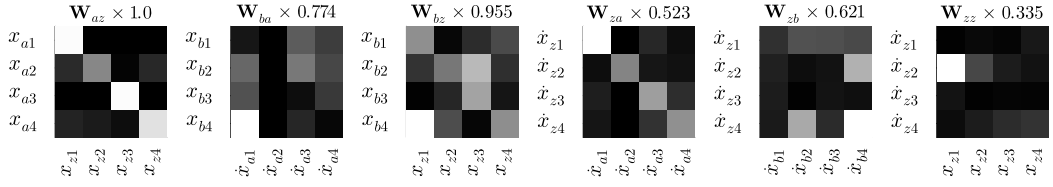


Figure 3.31: *PSDa* (3 training trajectories)

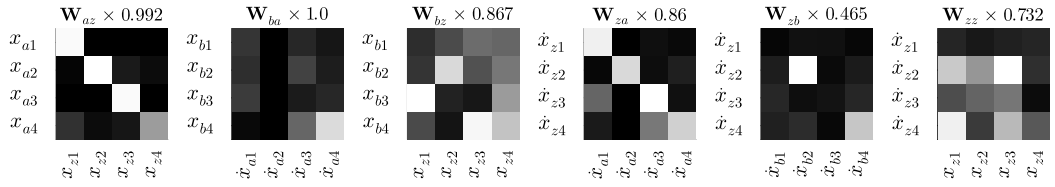


Figure 3.32: *PSDa* (4 training trajectories)

Investigating the *PSDa* architecture with increasing training data, starting from a single trajectory (s. Fig. 3.29) to four trajectories (s. Fig. 3.32), we find that some matrices change more than others. For example, the submatrices $\boldsymbol{W}_{az}$ (connecting the combined model state $\boldsymbol{x}_z$ with the subsystem state $\boldsymbol{x}_a$) and $\boldsymbol{W}_{za}$ (connecting the subsystem state derivative $\dot{\boldsymbol{x}}_a$ with the combined model state derivative $\dot{\boldsymbol{x}}_z$) stay close to the identity like shape they were initialized in. However, unexpected cross-correlations beyond the diagonal are present, too.

The matrix $\boldsymbol{W}_{bz}$ (connecting the combined model state $\boldsymbol{x}_z$ with the subsystem state $\boldsymbol{x}_b$) learns for a mixture of connection signals between the combined model state and the input layer of the ANN. We know, that the nonlinear friction is depending on the velocity of the bouncing ball. This means that the input of the ANN must be connected to a velocity signal to learn for the friction effect. This can be expressed in two ways: First, in the matrix $\boldsymbol{W}_{bz}$ by occupying the 2nd and 4th column (that correspond to the velocity within the state), or second, in the matrix $\boldsymbol{W}_{ba}$ by occupying the 1st and 3rd column (that correspond to the velocity within the state derivative). For any number of trajectories, we find that at least one of the two possibilities applies. Although connecting additional columns is not useful, non-causal but connected signals can still be filtered in the first layer of the ANN.

Within the matrix $\boldsymbol{W}_{zb}$ (connecting the submodel state derivative $\dot{\boldsymbol{x}}_z$ with the combined model state derivative $\dot{\boldsymbol{x}}_z$) we expect connections to extend the bouncing ball acceleration, so the entries for $\dot{\boldsymbol{x}}_{z2}$ (2nd row) and $\dot{\boldsymbol{x}}_{z4}$ (4th row) respectively. We observe, that these rows are well occupied and only very little non-causal signals in rows one and three are used. Finally, we expect the submatrix $\boldsymbol{W}_{zz}$ (directly connecting the combined system state $\boldsymbol{x}_z$ to its derivative $\dot{\boldsymbol{x}}_z$) to be zeros, however we find many learned connections, indicating that this model will not extrapolate well — as we already found out. However, the plots also show, that the number and strength of the wrongly learned correlations decreases with raising number of training trajectories — except in the case where the fourth trajectory is added.

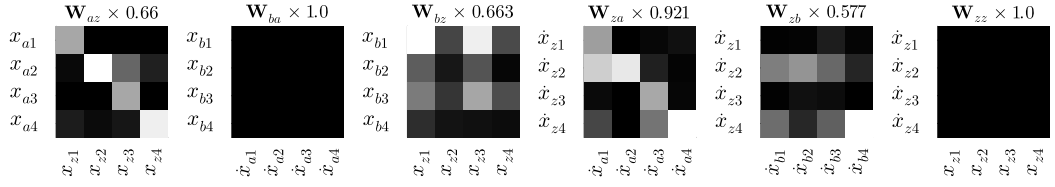


Figure 3.33: *P* (1 training trajectory)

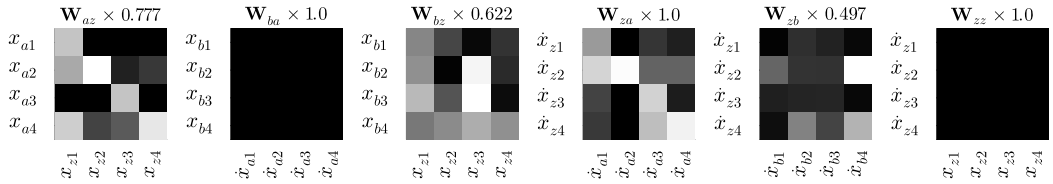


Figure 3.34: P (2 training trajectories)

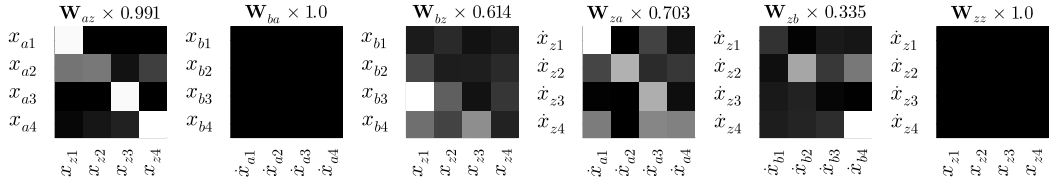


Figure 3.35: P (3 training trajectories)

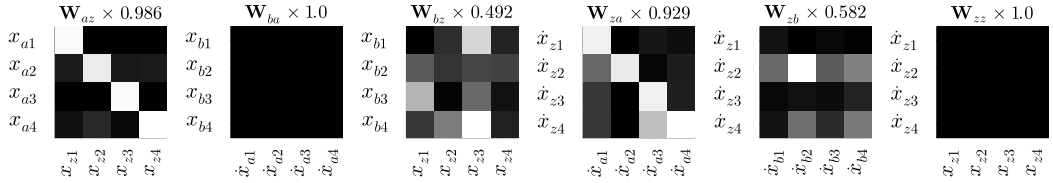


Figure 3.36: P (4 training trajectories)

Comparing these results to the ones of the architecture P, the most significant difference is that this architecture is not able to learn for the wrong correlations in submatrices $\boldsymbol{W}_{zz}$. Further, the serial connection by matrix $\boldsymbol{W}_{ba}$ is prohibited and forces the training of a strictly parallel pattern. In general, every submatrix is qualitatively closer to the expected training result. Worth mentioning, the submatrix $\boldsymbol{W}_{zb}$ (connecting the subsystem $\boldsymbol{s}_b$ state derivative to the combined model state derivative) is in good shape and further improves with increasing trajectory count.

Contribution 6 *By visualization and investigation of the learned connection matrix $\boldsymbol{W}$, we are able to evaluate how the submodels influence each other. This gives an additional opportunity to gain deeper insight into the explainability of hybrid models. We can even explore how submodels interact and gain knowledge on how a not modeled effect influences the (hybrid) model.*

Equipped with a new design pattern for generic and interpretable hybrid modeling, including a strategy for efficient event handling of hybrid models, methods for proper initialization of such models can be examined in the next chapter.

Chapter 4

Initialization

For every model featuring parameters, some initial parameter values are required. For physical models, these parameters are often determined by experiments (so measured) or known from development (e.g. CAD). For machine learning models, *initial values* are used to start an optimization which successively finds better suiting parameters. The topic of *initialization* deals with the appropriate selection of initial values — not only parameters — for hybrid models, to ensure efficient operation.

4.1 Introduction

We can interpret the term *initialization* from multiple perspectives:

Initial value From a mathematical perspective, solving of an ODE corresponds to an initial value problem (IVP). IVPs involve an (iterative) solving procedure, starting from an *initial value*. The same as for ODEs, for HUDA-ODEs the input to the solving algorithm is an *initial state* to start the first iteration of numerical integration. Finding an appropriate initial state before solving is referred to as *initialization*.

Numerical properties The hybrid model should be in *good condition* before being solved. The numerical conditioning of physical models, as well as ML models, heavily depends on the chosen parameterization. Because of interpretability, the parameters of the physical model are often fixed, or at least restricted to be within fixed boundaries or known distributions, whereas for ML models there are in general no such limitations or the limitations are driven more by numerical precision. Therefore, the appropriate parameterization of the involved submodels is crucial to obtain proper hybrid models.

Domain boundaries ML and physical models operate in different numerical ranges. Whereas physical models can generally operate with very small and very large numbers (technically only restricted by floating point issues), ML models are more sensitive, depending on the activation functions and depth of the architecture used (think of vanishing and exploding gradients). If combining ML and physical models, online pre- and post-processing of signals at the interfaces is therefore required, to fit the individual requirements by the different model types. The initialization of these processing operations needs investigation.

In summary, for initialization of hybrid models we need to distinguish between the initialization of physical and machine learning models, as well as the connections in between, which are all investigated individually in the following.

4.2 State of the art

In general, initialization becomes not researched for generic hybrid modeling approaches (to the knowledge of the author) and only use case-specific initialization routines are applied in the publications investigated in this thesis. As a consequence, we need to determine a suitable initialization process for HUDA-ODEs. However, the current state for initialization of *pure* physical and *pure* ML models can be briefly summarized and serve as a starting point for the derivation.

Physical models
Without further specification, the term *initialization* describes the search for initial values for solving the ODE in the domain of physical modeling. The initialization of physical models is well researched and can be seen as textbook knowledge. The solving of an initialization problem is even part of the FMI standard [41]. However, for challenging and specific application domains, there is still active research.

For physical models, the physical parameters are known in general or can at least be approximated within a meaningful range. If the parameters are not known[1], they can be identified using parameter estimation methods, for example, by gradient-based optimization as also used for ANNs. Parameters underlying the quality of production are often specified via a simple probability distribution, such as an expected value and a standard deviation.

ML models
In the world of ML, *initialization* refers in general not to finding initial values to solve an ODE — the majority of ML models do not solve an ODE at all — but to find initial *parameter* values to start optimization (*training*). Parameter initialization of ML models is a topic that is well studied, there are many publications available that introduce new and compare existing initialization routines for deep ANNs [89], [90], [91], to name only a few. Because initialization is crucial for the optimization success, newly introduced ML architectures are almost always presented together with a proper initialization strategy.

Interestingly, regarding known values, for ML models it is often the other way round compared to physical models: There is an intuitive understanding of the start state (think about the hidden state of an RNN), but it is not trivial to select reasonable parameter values to start the optimization. Especially in the context of UODEs, that require solving an ODE, initialization of these parameters is crucial. Without proper initialization, the eigenvalues of the system are unknown and the resulting ODE might be very stiff, which directly influences the training performance because solving requires more solver steps. Even worse, ODEs could be parameterized so that they are not even be solvable at all by the incorporated solver.

[1]Sometimes, also physical models include parameters that are accepted to be not (or at least hardly) interpretable. An example of this are complex friction models.

Open challenges
The open challenge for initialization of hybrid models in real-world engineering can be formulated straight forward: It is unclear how hybrid models (based on HUDA-ODEs) can be initialized for arbitrary application scenarios. This captures the initialization in terms of finding an appropriate initial state $\boldsymbol{x}_0$ (s. Sec. 4.3) and initial parameters $\boldsymbol{p}$ (s. Sec. 4.4), that promote an efficient training process. In addition to the parameters of the physical and ML model, hybrid modeling further introduces parameters for static and learnable connections (s. Sec. 4.5), that must be initialized, too.

4.3 Initial state

One of the aims of hybrid modeling is to maintain the physical explainability of the incorporated physical model. The explainability of the incorporated ML model shall even be increased. Consequently, the system state of the hybrid model is often *similar* to that of the physical model and, at least to a large extent, physically interpretable. For the initial state of a hybrid model, multiple cases for *initialization* can be distinguished:

Known The initial state for the hybrid model is known from data. Especially for systems with many states and/or only a few sensors, this is quite unrealistic. Even if the continuous part of the state vector is known in some applications, the discrete part is more unlikely to be known. This is because the continuous nature of physics itself: Most of the physical processes investigated happen continuously and are modeled discretely *only* for performance reasons. Therefore, the discrete part of the system is based on a design decision of the modeler, and not driven by the nature of physics. For example the bouncing ball colliding with the floor is a continuous, elastic process, but often — like in this thesis — modeled in a discontinuous way to avoid a very stiff equation system during contact. However, in the lucky case that the initial state is known, no further steps need to be applied and the IVP can be solved.

Optimized If the initial state is completely or partly unknown, the start state can be defined as parameter of the hybrid model and can be optimized together with other physical and/or ML parameters. If the start state is not unique in its entirety with regard to the available data, this procedure carries the risk of converging to a mathematically correct but physically inappropriate solution for the start state. This is often the case and more likely, the larger the investigated system becomes. In such cases, constraints can be applied to force the intended solution.

Stationarity For systems that repeat periodically over time — maybe after a phase of transient behavior — it can be assumed that the state converges to the correct state if initialized *close enough*. This is especially true if the system is initialized close to a stable equilibrium. In this case, the start state can be adopted iteratively by taking the last state of a period as new initial state for a new integration. This procedure is related to the *optimized* case above, but instead of optimizing in direction of the directional derivatives of the parameters, optimization is performed in the direction of the "average time derivative",

so the dynamic evolution of the system. This technique is successfully applied in Thummerer *et al.* [92] for the human cardiovascular system, that offers only a partly known state, but features the periodicity of the heart beat.

Determined If the initial state is not known or only in parts, an *initialization problem* can be defined to solve for the initial state $\boldsymbol{x}(t_0)$ based on variables like inputs, outputs, intermediate variables or incomplete parts of the state vector. This is the most common scenario in real world engineering and further investigated in the following. If an initialization routine for at least one of the submodels is already defined, as common for large physical simulation models, the hybrid model initialization needs to take this into account. Mathematically, this directly refers to the proper creation of the function $\boldsymbol{\alpha}$ within the HUDA-ODE, based on the corresponding initialization functions of the submodels to be combined.

Interestingly, finding a consistent start state for the hybrid model based on the start states of the physical and MLs models is strongly related to event handling for hybrid models: The problem statement behind both scenarios involves finding a global state that suits a determined local state. In general, there is no guarantee that the physical model and the ML model, each with their own initialization routines, calculate local start states that result in a consistent global start state for the hybrid model. Therefore, two cases for the initialization problem can be distinguished:

- If both subsystem feature states which are **completely independent** (for example if the global state is a concatenation of the subsystem state vectors), a consistent state is trivial to find, because both initialization routines can be performed individually. For example, if RNNs are paired with physical simulation models, the physical model and RNN do not share any states and can be initialized straightforward. Similarly, if only one submodel has a state (e.g., combination of physical model and FFNNs), the corresponding initialization routine within the submodel with state can be performed without complications. These cases make the choice for an initial state correspondingly trivial.

- If parts of the state are **shared** between submodels, one of the submodels, as during event handling, must be determined to dominate the initial state of the hybrid model. Otherwise, the initialization problem may not have a solution. Usually, the state of the physical model is rated higher in terms of explainability, so it also determines the initial state of the ML model.

To summarize, the general case is that states are independent (if both subsystems feature states) and uniquely correspond to a single subsystem. In such cases, the initialization problem can be solved to find an initial state for the hybrid model. We investigate only this case in the following. For other cases, individual solutions (similar to event handling) must be found, and no simple general statement is possible.

4.4 Initial parameters: ML and physical model

As already introduced, the initialization of deep ANNs is a complex topic and widely discussed. In addition, physical parameters are often known in advance or can be

determined experimentally. As a result, the parameters of hybrid models are often split in practice: There are strictly physical parameters for the physical model and "meaningless" parameters for the ML model. Even if parameters are shared between submodels (for example, physical parameters are selected as ANN input), the physical origin of the parameter usually dominates. This strict separation allows for easier initialization of parameters and motivates us to continue using the existing methods instead of deriving new methods for the parameter initialization of physical and ML models as part of hybrid models. Since the submodels within the hybrid model are only combined by the connection equations, these in turn can be initialized in such a way that the requirements of the individual parameter initialization routines of the submodels are not violated, which we further discuss in Sec. 4.5.

As a final note, in addition to the static initialization routines for ANNs, there exists an additional class of *pre-training* methods for NODEs with the same goal as for initialization: Provide the model with good initial conditions for optimal training. However, as the name suggests, pre-training involves gradient-based optimization and is therefore discussed as part of *training* in Cha. 5 and not as part of initialization.

4.5 Initial parameters: Connections

If parameterizable connections are used, as introduced in the previous chapter, these must also be initialized. In a way, the parameters of the connections exhibit properties of both the physical and the ML model: For some of the parameters, there is a very intuitive understanding, and the values are known beforehand. These are called *static connections*. For others, a value can only be assumed and should be identified by optimization, which are designated *learnable connections*. Similarly to the other parameters, initialization (so the occupancy of the matrix $\boldsymbol{W}$) of connections between physical and ML model has a crucial impact on different attributes of the hybrid model:

Generalizability If only very little data is available and non-causal, but correlating signals[2] are connected, the generalizability of the machine learning model, and therefore also the resulting hybrid model, suffers because of learning wrong causalities based on data correlations. However, if *enough* data is available, so that all causalities can be correctly identified, all possible signals can be connected to allow the identification of the relevant ones. The resulting hybrid model can be examined after training, e.g. using local sensitivity analysis, and the connection architecture can be optimized based on the impact of the signals, where signals with no or low impact are cut.

Performance The number of connected signals has a direct impact on training performance. The more signals are connected, the more sensitivities need to be determined. Connecting non-causal signals therefore unnecessarily worsens the overall training performance. Even if the impact is less significant during inference, because no sensitivities are determined, the problem of reduced performance also applies here.

[2]Signals that do not causally influence the loss function, but correlate with it.

This observation sheds light on the trade-off between the degrees of freedom for the connections and the available data. Even if in theory one could keep the hybrid model structure generic, the practical application motivates the usage of an optimized and use case-specific architecture, if resources (data, computational power) is limited. Examples for common architectures are given in Sec. 7.2. Static connections can be initialized with the signal operators presented in Sec. 3.4.2. Learnable connections should be initialized with noisy[3] 0 or 1, depending on the assumed connectivity.

In the following, two specific building blocks that are fully expressed by the connection parameters within the proposed hybrid model architecture are presented.

4.5.1 Pre- and post-processing

As applied in multiple publications [83], [92], [93], we propose to reuse the well-studied and established initialization methods for the physical model state as well as the ML model parameters. This reduces the actual problem to the initialization of the connections so that the numerical ranges for the physical and ML model are not violated: Whereas the physical model expects values in a predefined and physically meaningful range, it is good practice to provide pre-processed data to deep ANNs that is either standardized (for normal distributed data) or normalized (for non-normal distributed data). Existing initializations can be preserved using the following linear *pre-processing* operation[4]:

$$\boldsymbol{c}_{pr}(\boldsymbol{v}) = \boldsymbol{v} \odot \boldsymbol{w}_{pr} + \boldsymbol{b}_{pr}. \tag{4.1}$$

In analogy to pre-processing, a *post-processing* operation can be defined:

$$\boldsymbol{c}_{po}(\boldsymbol{v}) = \boldsymbol{v} \odot \boldsymbol{w}_{po} + \boldsymbol{b}_{po}. \tag{4.2}$$

Neglecting pre- and post-processing leads to drastically loss of expressiveness. This can be *saturation*, e.g., hyperbolic tangent and sigmoid saturate to 1 for large values, or the actual nonlinearity can be wasted, e.g. hyperbolic tangent and sigmoid are almost linear around 0 for very small perturbations.

The parameters of pre- ($\boldsymbol{w}_{pr}$ and $\boldsymbol{b}_{pr}$) and post-processing ($\boldsymbol{w}_{po}$ and $\boldsymbol{b}_{po}$) can be used in different ways:

- If there is a good understanding regarding data and the quantities that are computed by the ANN, the parameters for pre- and post-processing can be determined individually and applied in a **static** way, so without further adapting them.
- In applications where the ANN operates as *corrector*, therefore having the same interpretation for input and output signals, the two operations can be **paired**. Note that for $\boldsymbol{w}_{po} = \boldsymbol{1} \oslash \boldsymbol{w}_{pr}$ and $\boldsymbol{b}_{po} = -\boldsymbol{b}_{pr} \oslash \boldsymbol{w}_{pr}$ it states that both

[3]Initialization with exact 0 and 1 could cause that some parameters have the same sensitivities and will therefore be changed identically (depending on the optimizer), which is not intended. In practice, uniform noise within $\pm 1e-3$ was often applied, but the type and exact quantity (within a meaningful range) of noise seems to only have little impact in combination with non-deep ANNs.

[4]This is also implemented in the `ShiftScale` and `ScaleShift` layers within *FMIFlux.jl*.

operations for pre- and post-processing are inverses of each other, so $\boldsymbol{c}_{pr}^{-1} = \boldsymbol{c}_{po}$ and $\boldsymbol{c}_{pr} = \boldsymbol{c}_{po}^{-1}$, resulting in $\boldsymbol{c}_{po}(\boldsymbol{c}_{pr}(\boldsymbol{v})) = \boldsymbol{v}$. This can be used to initialize processing layers that belong to each other, thus forming a transformation shell around the ANN. In such cases, two operation modes can be distinguished:

- As mentioned, the parameters of one of the processing layers can be initialized statically or ...
- ... the parameters of one of the processing layers can be optimized to adapt to new signal ranges which might occur during training progress, ...

... while the parameters for the *other* processing layer are fully determined by computing the inverse.

It is common in modeling deep ANNs, that the last layer does not use an activation function (or identity, respectively) to allow for output values beyond the output range of the activation. This makes it appear that the usage of a dedicated post-processing layer is obsolete. However, applying a proper post-processing operation is meaningful for the following reasons:

- Especially for large output values ($> 10^3$), the ANN needs a significant number of training steps to learn for a large scaling in the last layer. This is often unnecessary if the amount of scaling is (roughly) known beforehand.
- The last layer of the ANN can use a nonlinear activation function, which improves nonlinear expressiveness, especially for shallow (not deep) ANNs without introducing additional layers/parameters. If the connection after the ANN (the post-processing layer respectively) holds trainable parameters, this is even recommended, because otherwise the last linear layer and the following linear connection collapse, resulting in a single linear operation with linear dependent parameters — which again can hinder proper optimization.

In summary, the mathematics behind pre- and post-processing is quite trivial, but it is one of the most important steps in hybrid modeling and has a crucial influence on training success, as we will show in an experiment (s. Sec. 4.6.1).

4.5.2 Gates

The challenge of initialization can be bypassed or at least strongly attenuated by applying *gates* as introduced in Thummerer *et al.* [83]. Here, the term *gate* refers to a specific parameter within one of the connect equations, that is changed by optimization or a control strategy during model training. Introducing gates therefore corresponds to declaring some connection parameters tunable. This enables for a very intuitive control, surveillance, and analysis of the training process and therefore enhances *explainability*. Gates can further be used to avoid initialization by reducing the influence of bad initialized submodels (or parts of them) within the hybrid model, by closing a gate (almost) completely. The idea is, that as long as the impact of the bad initialization is small, parameters can be optimized in a better direction because the resulting model is still solvable. For example, if a high fidelity physical model is present as part of the hybrid model and only minor adjustments are expected to be made by an UA, we can introduce a specific gate that reduces the influence

of the UA to (almost) zero for the training start, compare Fig. 4.1. With further training progress, the gate, respectively the influence of the UA, can be opened in a controlled way (e.g. based on the number of training steps or proceeding convergence) or optimized together with other parameters.

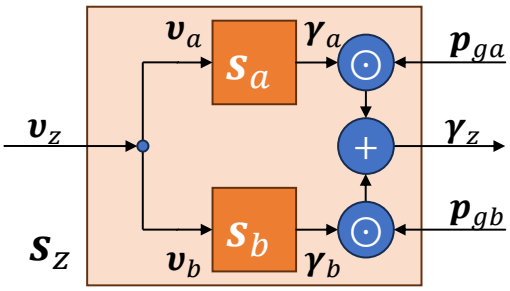


Figure 4.1: An example for applying *gates*: The input $\boldsymbol{v}_z$ is shared between both subsystem, that calculate outputs $\boldsymbol{\gamma}_a$ and $\boldsymbol{\gamma}_b$. After passing the gates, that are parameterized by $\boldsymbol{p}_{ga}$ and $\boldsymbol{p}_{gb}$, the results are summed up to obtain the hybrid model output $\boldsymbol{\gamma}_z$.

Note by comparing this to the *PSD* architecture or one of its derivatives, such as for example the *P* architecture in Fig. 3.22, that such gates can be implemented by the connect equation $\boldsymbol{c}_z$:

$$\boldsymbol{\gamma}_z = \boldsymbol{c}_z(\boldsymbol{\gamma}_a, \boldsymbol{\gamma}_b, \boldsymbol{p}_{ga}, \boldsymbol{p}_{gb}) = \boldsymbol{p}_{ga} \odot \boldsymbol{\gamma}_a + \boldsymbol{p}_{gb} \odot \boldsymbol{\gamma}_b. \tag{4.3}$$

We recall that $\odot$ stands for the Hadamard product, so element-wise multiplication of the vector elements. Because this corresponds to a linear equation, we find that this equation can be fully expressed by the generic connect equation layout that is already present within the *PSD* architecture:

$$\boldsymbol{p}_{ga} \odot \boldsymbol{\gamma}_a + \boldsymbol{p}_{gb} \odot \boldsymbol{\gamma}_b = \boldsymbol{W}_a \cdot \boldsymbol{\gamma}_a + \boldsymbol{W}_b \cdot \boldsymbol{\gamma}_b + \boldsymbol{b}. \tag{4.4}$$

Here, $\boldsymbol{W}_a$ and $\boldsymbol{W}_b$ are submatrices of $\boldsymbol{W}$ and $\boldsymbol{W}_a = diag(\boldsymbol{p}_{ga})$, $\boldsymbol{W}_b = diag(\boldsymbol{p}_{gb})$, and $\boldsymbol{b} = \boldsymbol{0}$. The diagonals of $\boldsymbol{W}_a$ and $\boldsymbol{W}_b$ now correspond to the (trainable) gate parameters. It is important to clarify that besides $\boldsymbol{c}_z$, any of the connect equations can be used to express gates for any connection.

Depending on the values for $\boldsymbol{p}_{ga}$ and $\boldsymbol{p}_{gb}$, different cases can be investigated:

- For $\boldsymbol{p}_{ga} = \boldsymbol{1} \wedge \boldsymbol{p}_{gb} = \boldsymbol{0}$, the subsystem $\boldsymbol{s}_a$ fully determines the output of the hybrid model, so $\boldsymbol{\gamma}_z = \boldsymbol{\gamma}_a$. This is useful if $\boldsymbol{s}_b$ is not properly initialized and $\boldsymbol{s}_a$ is known to provide a good solution. For $\boldsymbol{p}_{ga} = \boldsymbol{0} \wedge \boldsymbol{p}_{gb} = \boldsymbol{1}$, it is exactly the opposite.
- For $\boldsymbol{p}_{ga} = \boldsymbol{1} \wedge \boldsymbol{p}_{gb} = \boldsymbol{1}$, the output of the hybrid model is equal to the sum of the outputs of the subsystem, so $\boldsymbol{\gamma}_z = \boldsymbol{\gamma}_a + \boldsymbol{\gamma}_b$.
- For $\boldsymbol{p}_{ga} = \boldsymbol{0} \wedge \boldsymbol{p}_{gb} = \boldsymbol{0}$, the output of the hybrid model is zero, which can be used if both subsystems are poorly initialized.

In addition to that, the gate parameters are interpretable in the same way as any other parameter of the matrix $\boldsymbol{W}$, as investigated in Sec. 3.4.2. Gates can be used to control not only signals, but also the entire architecture layout. This is exemplified and investigated in Sec. 7.2.5. Finally, the benefits of using gates are summarized in an experiment, see Sec. 4.6.2.

4.6 Experiments

We illustrate the findings in this chapter using two experiments as examples.

4.6.1 Pre- and post-processing

For the pre- and post-processing experiment, we deploy a hybrid model and examine the impact of different pre- and post-processing setups. The sequential architecture to process dynamics (s. Sec. 7.2.1) is applied for the hybrid model, where the subsystem $\boldsymbol{s}_a$ is the bouncing ball as introduced in Sec. 2.4, but without air friction, and $\boldsymbol{s}_b$ is a FFNN. The goal for the hybrid model is to learn for the neglected friction effect. The boundary conditions for the experiment are given in Tab. 4.1. In addition, we start a dedicated experiment for every permutation of *optimized parameters* (O) and *paired pre- and post-processing* (P), which leads to the following configurations:

$\sim$ No pre- and post-processing (baseline).

O The parameters of the pre- and post-processing layers are optimized individually.

P Both layers are paired, so the post-processing layer is the inverse of the transformation of the pre-processing layer, but the parameters are not optimized. The pre-processing layer is initialized so that pre-processed data features a standard deviation (SD) of 1 and mean of 0.

PO The pre- and post-processing layers are paired, whereas the parameters for the pre-processing layer are optimized, and the post-processing layer is parameterized based on the inverse transformation of the pre-processing layer.

Table 4.1: Setup for the pre- and post-processing experiment.

Attribute	Value
Training strategy	Growing Horizon (SB)
Horizon start	0.25 s
Horizon loss threshold	0.05
Horizon increment	0.01 s
Horizon length	3.0 s
Loss function	MAE (s_x, v_x, s_y, v_y)
Loss evaluation points	$\{0.0\,\text{s}, 0.01\,\text{s}, ..., 2.99\,\text{s}, 3.0\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam ($\eta = 0.001$)
Architecture	Sa (SOP)
ANN layout	$4 \rightarrow 8$ (tanh)
	$8 \rightarrow 8$ (tanh)
	$8 \rightarrow 2$ (identity)
Number of training steps	2,000
Number of repetitions	100

4.6.1.1 Hypothesis

The option to optimize the pre- and post-processing operations (mode O) together with the actual ML model seems appealing at first glance, because optimization of pre- and post-processing allows to flexibly react to new signal ranges. We assume that this approach should be able to identify the required transformation values. Less intuitive is if the *pairing* (P) of the pre- and post-processing operation leads to faster convergence or even the same quantitative minimum. The same applies to the case where both modes are used simultaneously (PO).

4.6.1.2 Results & discussion

The first important observation is that for the investigated example, all examined cases for pre- and post-processing converge to similar minima, whereas not applying proper pre- and post-processing ($\sim$) is not able to converge to an acceptable minimum, see Fig. 4.2. This first observation highlights the importance of the proposed measure in general.

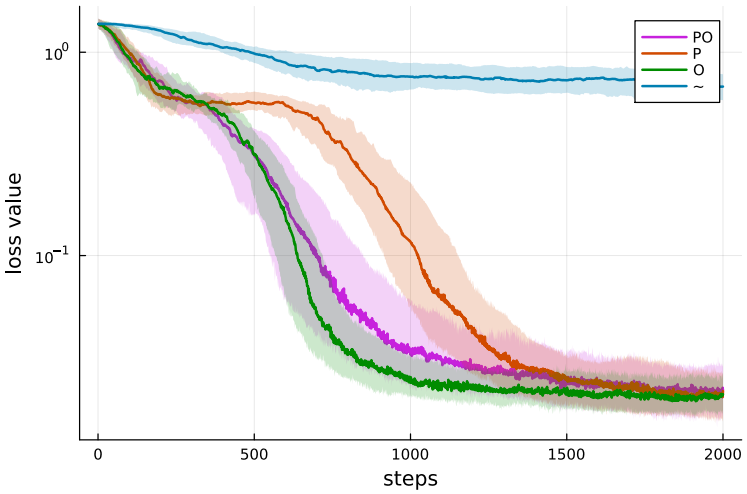
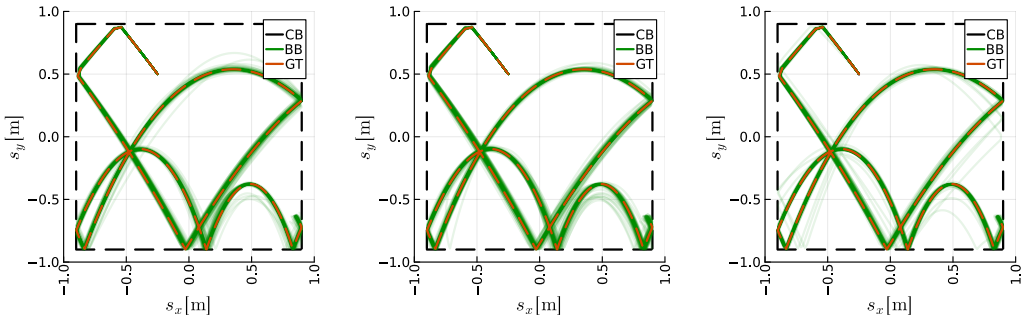


Figure 4.2: Loss curves (median) over time, comparing the different pre- and post-processing operation modes. The semi-transparent ribbons show the 25th and 75th percentiles.

Further, we observe that in the considered case the methods involving the optimized approach (O and PO) feature the fastest convergence, followed by P. We can explain the slower convergence for P by the initialization strategy for the pre-processing and the resulting post-processing operation, which are based on the mean and SD of the corresponding signals. This results in a suboptimal operation range for the hyperbolic tangent activation function.

Due to the small loss value, we also expect a good qualitative fit of the trajectories, which are shown in Fig. 4.3. All plots show a similarly sharp convergence to the target trajectory, apart from a few outliers.

This observation can be quantitatively verified by investigating the loss values in Tab. 4.2. We find that O not only converges the fastest (by investigating step $1,000$), but also converges to the best median optimum (by investigating step $2,000$) and features the smallest P_{75} of various runs. Interestingly, the P_{25} is similar across



(a) P (b) PO (c) O

Figure 4.3: Comparison of the training simulation trajectory of the hybrid model using different pre- and post-processing operations. Multiple trajectories are generated by repetition of the experiment with changing ANN initialization.

all runs, which means that in applications with high repetition rates (and selection of the best run) all approaches can in principle deliver good results.

Table 4.2: Quantitative results for the pre- and post-processing experiment: Median, 25th and 75th percentiles, and minimum of the loss values.

Experiment	Step	Median	P_{25} | P_{75}	Min
PO	1,000	0.032621	[0.021983, 0.058974]	0.010117
P	1,000	0.117071	[0.057121, 0.230749]	0.013034
O	1,000	0.025027	[0.020125, 0.032296]	0.01333
~	1,000	0.75818	[0.655461, 0.884976]	0.46157
PO	2,000	0.021751	[0.015765, 0.028645]	0.007084
P	2,000	0.021564	[0.015848, 0.026652]	0.006503
O	2,000	0.02082	[0.017026, 0.025823]	0.010707
~	2,000	0.677971	[0.581015, 0.78002]	0.297191

To conclude, besides the theoretical considerations, optimization of pre- and post-processing parameters is a valuable measure for the considered example, which motivates the use of this measure for hybrid modeling in general. It is important to mention that the importance of this step increases massively when numerically larger ranges are considered. Compared to the approach P, which does not optimize the parameters, convergence is significantly faster for the optimized approaches. If fewer parameters are required, the optimized (O) approach can be combined with pairing (PO), which still features fast convergence to an appropriate minimum, but without introducing additional parameters for a second transformation. Further, it is important to state that in real life applications, the selection of static parameters for pre- and post-processing is not an obvious choice. For example, imposing a mean of 0 and SD of 1 is not suitable for every application, because this leads to a drastic saturation of outliers by the activation functions within the ANN, which is often unacceptable for physical simulation models. In such cases, the approaches involving optimization (O and PO) provide a valuable solution. However, further — and possibly application specific — strategies for finding initial values for the processing

operations, independent of if they are later optimized or not, are implementable and meaningful, e.g. scaling and shifting to a given range.

Contribution 7 *We found that pre- and post-processing at domain boundaries (between physical and ML model) is necessary for proper training of hybrid models. We showed that all operation modes improve training convergence, however, optimizing the parameters is a valuable measure if the value ranges at the interfaces are unknown or change significantly during training. Linear pre- and post-processing operations can be expressed by the connection matrix* $\boldsymbol{W}$ *and bias* $\boldsymbol{b}$ *within the proposed* PSD *architecture (and derived architectures), because here, connections are modeled as linear and parameterizable functions.*

4.6.2 Gates

For the gates experiment, we want to investigate how different *operation modes* for gates influence the training of a hybrid model. The parallel architecture to add dynamics (s. Sec. 7.2.4) is applied, where the subsystem $\boldsymbol{s}_a$ is the bouncing ball as introduced in Sec. 2.4, but without air friction, and $\boldsymbol{s}_b$ is a FFNN. The goal for the hybrid model is to learn for the neglected friction effect. Three different types of gate *operation modes* are compared:

Static (S): The gates are not changed and are kept *open* (so $p_{ax} = p_{ay} = 1$). This corresponds to not using gates at all, but incorporating a straight connection.

Controlled (C): The gates are driven linearly (w.r.t. the training steps) from 0 to 1, so that after 50 % of the training the gates are fully open. For the second half of the training, they remain open.

Optimized (O): The gates are initialized with zeros (so fully closed, $p_{ax} = p_{ay} = 0$) and are optimized together with the parameters of the FFNN.

Detailed information on the experiment can be found in Tab. 4.3.

4.6.2.1 Hypothesis

Intuitively, we can argue that the *optimized* approach should lead to good results, because the optimizer can adapt to an *optimal* opening of the gates so that the parameters within the (perhaps badly initialized) FFNN can be corrected. The *static* approach may be the most cumbersome, because the ODE solver needs to deal with systems featuring random dynamics, that might be stiff and/or even unstable. This *not initialized* state needs to be recovered during the first optimization steps. This suboptimal start for optimization could be an advantage for the *controlled* strategy, because at the beginning of the training the influence of the (maybe inadequately initialized) FFNN is zero and even after some training steps it is quite small. The optimizer can make necessary corrections to the FFNN parameters during the first optimization steps.

4.6.2.2 Results & discussion

After performing the experiment, we start by investigating the progression of the loss curve, see Fig. 4.4.

Table 4.3: Setup for the gates experiment.

Attribute	Value
Training strategy	Growing Horizon (SB)
Horizon start	0.25 s
Horizon loss threshold	0.05
Horizon increment	0.01 s
Horizon length	3.0 s
Loss function	MAE (s_x, v_x, s_y, v_y)
Loss evaluation points	$\{0.0\,\text{s}, 0.01\,\text{s}, ..., 2.99\,\text{s}, 3.0\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam ($\eta = 0.001$)
Architecture	P (SOP, with gates)
ANN layout	$4 \to 8$ (tanh)
	$8 \to 8$ (tanh)
	$8 \to 2$ (identity)
Number of training steps	5,000
Number of repetitions	100

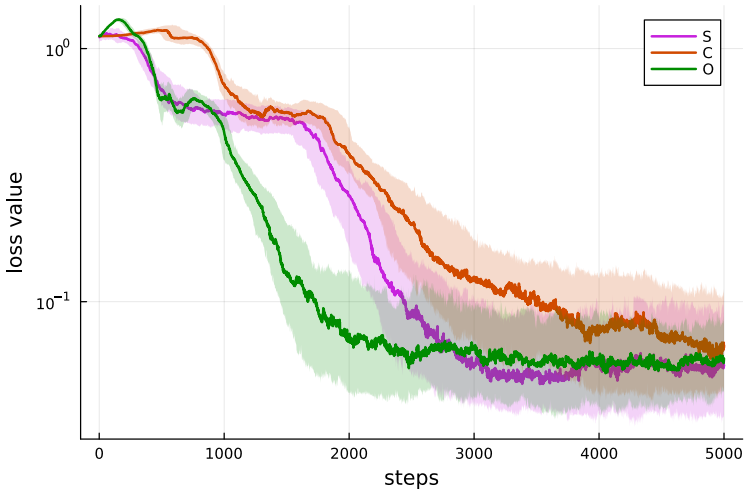


Figure 4.4: Losses (median) over training, comparing different gate operation modes. The semi-transparent ribbons show the 25th and 75th percentiles.

In Fig. 4.4, we can observe that at the very beginning the optimized approach converges the fastest, followed by the static approach. Whereas the fast convergence of the optimized approach mirrors our hypothesis, the static approach performs better than expected, the influences of a possibly poor initialization in combination with a static gate opening are obviously less than expected in the bouncing ball example. All approaches converge to similar minima, whereas the static approach converges to the best optimum (quantitatively).

The controlled and static approaches need significantly longer to converge, which is due to the suboptimal fixed values for the gates (for the controlled approach the selected slope, for the static approach the fixed value). As can be seen in Fig. 4.5, the optimized approach chooses a significantly steeper slope to control the gates, compared to the statically chosen slope for the controlled approach.

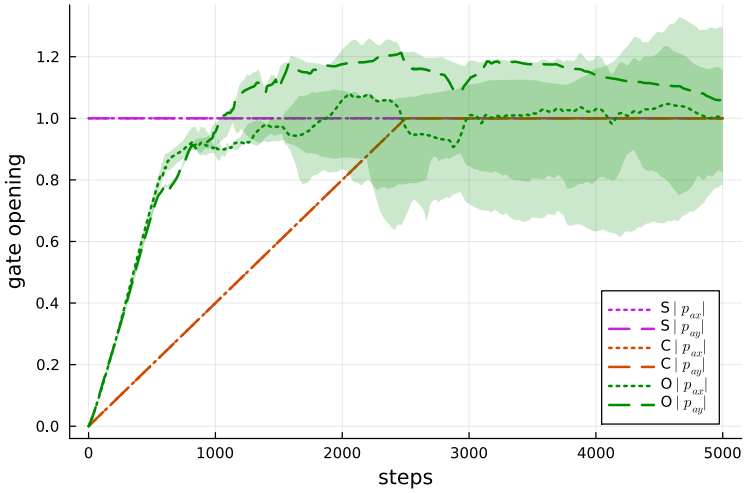
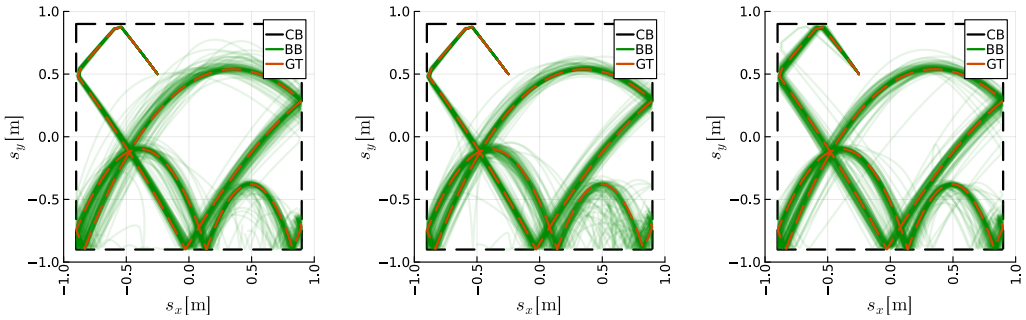


Figure 4.5: Opening of the acceleration gates $|p_{ax}|$ and $|p_{ay}|$ over the training for the different operation strategies. The semi-transparent ribbons show the 25th and 75th percentiles.

In further investigation of the opening of the gates, we find the expected opening strategy for the case C, as well as the static opening for the case S. More interesting is, of course, the opening for the optimized gates (O): Toward step $1,000$, both gates for acceleration in x and y are almost completely opened. During further training progress, the typical gate (25th and 75th percentiles) converges to values within $[0.8, 1.2]$, while the scattering (shown by the percentiles) increases first and then stagnates with a roughly constant deviation. The optimized openings are not exactly 1 because the first layer of the FFNN and gates both start by evaluation of a linear equation, and therefore these parameters in the hybrid model are linearly dependent — so can be used interchangeably.

Finally, investigating the actual trajectories of the ball (s. Fig. 4.6), we find a qualitatively nice fit between all methods on average, whereas the mode C visually shows a little more variance. Compared to the quantitative results in Tab. 4.4 that confirm this assumption, all methods show a small final error (median and percentiles), with increased values for the percentiles of the C approach. Interestingly, the static approach features the smallest median and the smallest 25th percentile after full training. One explanation for this is that the gate values are very sensitive (small changes have significant influence on the system dynamics), and the optimiza-



(a) S (b) C (c) O

Figure 4.6: Comparison of the different gate operation strategies by plotting the simulation trajectory of the hybrid model (HM) of each repetition (spaghetti plot), together with the ground truth (GT) and the collision border (CB).

tion of these values therefore has a higher affinity to instability of the optimization. The static approach is the only one that does not suffer from this risk. However, it is important to recognize that this is only the case for investigation of the final loss value (step $5,000$), in general, the static approach converges much slower compared to the optimized approach.

Table 4.4: Quantitative results for the gates experiment: Median, 25th and 75th percentiles, and minimum of the loss values of all repetitions.

Experiment	Step	Median	P_{25} | P_{75}	Min
S	1,000	0.549955	[0.479857, 0.631672]	0.366308
C	1,000	0.718248	[0.631637, 0.802786]	0.463922
O	1,000	0.465893	[0.361276, 0.539407]	0.194046
S	2,000	0.262257	[0.166608, 0.355698]	0.030938
C	2,000	0.383092	[0.330982, 0.487806]	0.131429
O	2,000	0.072003	[0.043945, 0.13323]	0.023002
S	5,000	0.055764	[0.034747, 0.105229]	0.020423
C	5,000	0.067306	[0.045272, 0.108774]	0.015961
O	5,000	0.059661	[0.044049, 0.087886]	0.016815

To summarize, manually controlling gates seems only meaningful for very specific applications, e.g. when a well-suited slope for the gate openings is known beforehand. The optimized approach converges significantly faster than the static approach. However, the static approach can deliver slightly more accurate results when fully converged. Together with the benefit that further initialization is not required, optimizing gates seems to be a valuable method for hybrid model initialization. As a final note, initialization can also be bypassed by applying pre-training by collocation as introduced in Sec. 5.4. Here, the HUDA-ODE is not solved at all, and therefore side effects of numerical solving such as stiffness and instability cannot appear.

Contribution 8 *We found that adding* gates *for signals within the hybrid model offers an elegant way to bypass the initialization problem. Gates offer a valuable tool for keeping parts of the architecture — starting from individual signals to the entire layout — flexible, while also increasing explainability. They provide a very well interpretable add-on to the* PSD *architecture and are technically fully contained within it, so that there is no additional implementation or computation overhead. Gate parameters can be initialized and optimized (during training) or be controlled by additional strategies.*

Chapter 5

Training

If we try to train a sophisticated hybrid model with a standard optimization loop, it usually converges to an unsatisfactory local minimum quickly. In this chapter, we investigate why and what we can do about it.

5.1 Introduction

Training of HUDA-ODEs is closely related to training of neural ODEs. Training a neural ODE involves three steps: First, a chunk of data is selected for gradient computation. Second, the parameter gradient of the loss function, which involves differentiation of an ODE solution, is determined. And third, the parameters are modified based on the gradient by an optimization algorithm.

The selection of data has obviously a significant influence on the success of the training. Different sampling strategies can be applied, examples for *scheduling* algorithms are given in Sec. A.4. Regarding gradient determination, we find that methods for SA that work for event ODEs (s. Sec. 2.1.3) can be reused for HUDA-ODEs as well, technically we derived the HUDA-ODE on basis of a UODE. And, of course, because training a HUDA-ODE fits the pattern of a classical optimization problem, existing optimizers can be applied[1].

5.2 Related work

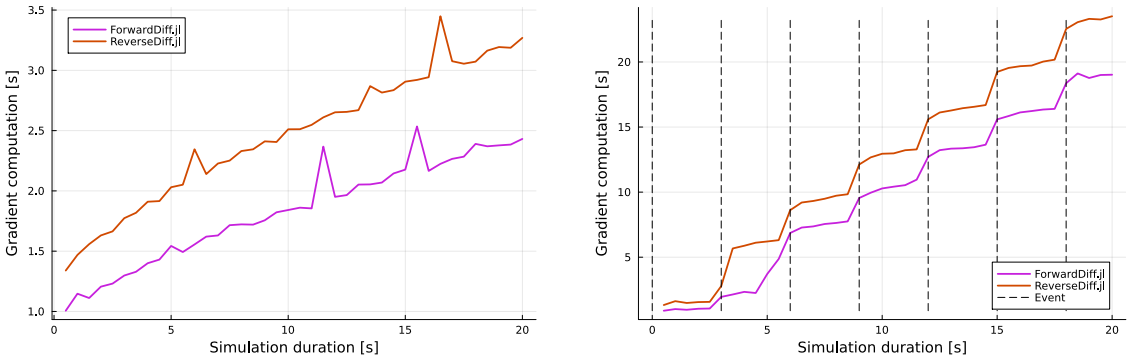
Since there is of course no state of the art for the training of HUDA-ODEs, common methods for training NODEs are introduced below and their applicability to HUDA-ODEs is discussed.

Single shooting
Just like NODEs, solving of HUDA-ODEs falls into the class of IVPs, i.e. a start state must be determined in order to iteratively solve the problem. The initial state can be given or be determined applying the methods for initialization discussed in Sec. 4.3. Based on an initial state, the problem is solved and a solution over time (trajectory) is obtained. Because the solved trajectory, during inference as well as during training, depends solely on the initial state and the parameters of

[1]However, new and specialized optimizers might lead to better results, as briefly investigated in the outlook.

the system, training on the entire solution trajectory is called *single shooting* [94], [95]. As a result, single shooting is only applicable if the data consists of a *single* measurement trajectory as well. If multiple trajectories are present (for example because of varying initial values, inputs, or parameterization), single shooting can only be applied to one trajectory at a time without methodological adjustments.

Briefly investigating computational performance, the cost for the gradient depends (for a constant stiffness problem) linearly on the simulation time span, see Fig. 5.1. However, in practice, there is a small computational overhead: If the stiffness of the ODE remains (almost) constant, there is nevertheless a short *warm-up*, in which the correct step size is determined by the solver step size control, starting with a sufficiently small step, resulting in a total computational cost above linear w.r.t. wall-clock-time. After that, the number of steps is approximately linear compared to the time span simulated. However, similarly to solving the first steps, at event instances the solver step size needs to be reduced to the initial value and slowly increases again.



(a) Without events.

(b) Periodic time events with 1/3 Hz.

Figure 5.1: Computation time for SA over an ODE solution, featuring 2 states and 50 parameters. The AD libraries *ForwardDiff.jl* (forward-mode) and *ReverseDiff.jl* (reverse-mode) are compared. Note that the ODE is linear and therefore has constant stiffness. Plots show the median performance of > 100 repetitions.

Anyway, training on long data sections (entire trajectories) is often impractical, this is because of:

Vanishing information Longer simulation intervals result in more solver steps being performed. The more steps the solver performs, the less is the sensitivity information of the individual solver step weighted in the finally obtained gradient. This naturally leads to a preference by the ANN for low frequencies (which are present along more solver steps) or the learning of a simplified trajectory that is missing details (underfitting), respectively.

Rising risks Considering a larger amount of data, it is very unlikely that the model is numerically well conditioned on each possible data section at each progress point (parameter set) of the training. The longer the interval trained on *at once*, the greater the risk that the current parameterization for the hybrid model leads to an unsolvable (or hardly solvable) problem. Working on an incomplete solution is not always feasible. For example, for continuous adjoint

methods, the original system might be solvable, but the adjoint problem might not, leading to termination of the gradient determination.

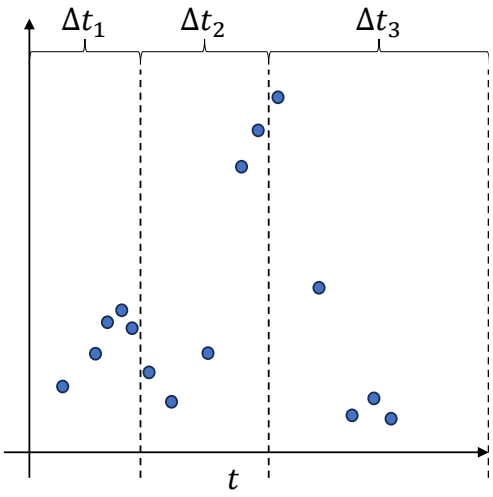
These observations motivate the investigation of training strategies that do not operate on the entire trajectory at once, but on smaller *chunks* of it.

Growing horizon
To address the loss (or strong averaging) of gradient information leading to underfitting, a common approach is the *growing training horizon* as used in Bruder and Mikelsons [96]. The training is initially performed on a subpart, starting at the beginning of the trajectory, and the training horizon is successively extended with increasing convergence, e.g., defined by a threshold for the cost function. This effectively prevents the training from converging to an unintended local minimum at the very beginning. However, this procedure requires the identification of sensitive hyperparameters for the start section length, the loss function threshold, the strategy for the extension of the training section, and the actual amount of the extension that is performed. In addition, the cost of training on the full horizon is as high as in single shooting, which might be infeasible in the real application. However, this training strategy does not require knowledge of the state besides for t_0 (initial state) and is therefore applicable for applications with partly known state trajectories.

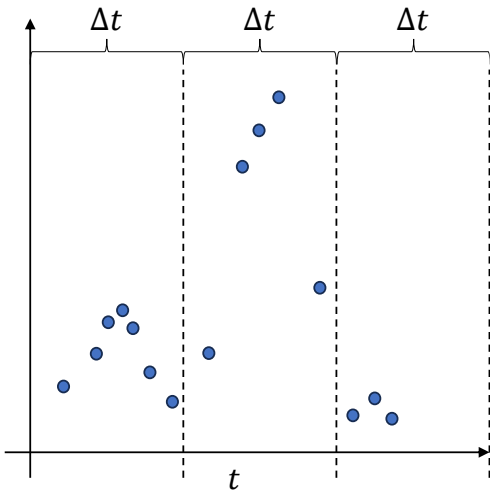
(Mini-) Batching
In ML, the term *sample* refers to a piece of input data that is fed into a model at once. Often in supervised learning, *samples* are further associated with the corresponding output data that is expected from the model. The term *mini-batching* describes the subdivision of data into groups of samples that are referred to as *mini-batches*. In the following, we omit the prefix *mini*, but always refer to e.g. mini-batching or mini-batches. One batch is picked and processed entirely, meaning the gradient is determined for the entire batch at once. The *batch size* identifies how many samples are grouped in a single batch. The selection of the next batch to perform a training step is performed by a *scheduler*. More information on the investigated scheduling algorithms for hybrid modeling, e.g. as part of the *JuliaCon2023* workshop [97], can be found in Sec. A.4.

For time series, one can think of multiple ways to batch. In this work, a sample is referred to as data of a single point in time, including states, inputs, outputs, and time itself. As a result, a batch refers to multiple, consecutive samples — so a piece of data over time. Instead of *batch size*, which refers to a number of samples, a more intuitive *batch duration* (s. Fig. 5.2) is used, that identifies a time span. This makes the batches independent of the sampling of data records (or the number of ODE solver steps, respectively) but has the disadvantage that different batches may exhibit different execution times. However, for systems with varying stiffness, this is the case anyway. Alternatively, dynamic-length batch elements can be created during training (*active learning*, see outlook Sec. 9.3).

Multiple shooting
Another approach, which executes multiple batch elements in parallel before determining the gradient, is *multiple shooting* [95]. Because multiple gradients are determined in parallel, a strategy is required to obtain a single gradient for common



(a) Conventionally, data is batched with a *batch size*, meaning a fixed size of data points. Note, that the number of samples is identically between batches, but the time span varies, so $\Delta t_1 \neq \Delta t_2 \neq \Delta t_3$.



(b) In this thesis, data (blue) is batched with the *batch duration* Δt. Note, that the number of included samples varies between the batches, but the time span is identically Δt for all (mini-) batches.

Figure 5.2: Comparison of different ways to batch a time series.

single-objective optimization. Often, the gradients are summed up or the mean of the gradients is used. Multiple shooting allows for better convergence especially for problems that tend to under-fit. This becomes more intuitive if we recall the issue of *vanishing information* for long solution trajectories.

Batching in general, as well as multiple shooting, features a problem that is not known from single shooting: Discontinuities at the boundaries between two consecutive batch elements, referred to as *shooting gaps* in multiple shooting, see Fig. 5.3. These gaps must be closed by additional optimization constraints. For example, the deviation of the last point of the trajectory, which defines the size of the shooting gap, can be penalized disproportionately. Although it has been shown that compliance with these constraints can be implemented by adding a regularization term to the cost function, applying the *augmented Lagrangian method* has also been shown to be successful [95].

In fact, batching and multiple shooting are very similar — one can think of multiple shooting as parallelized batching method — and the boundaries vanish, if batching is combined with an optimizer with *memory* that smoothes the gradient, like Adam [98]. These observations are summarized in Fig. 5.4.

In addition to the fact that batching and multiple shooting often result in better training performance compared to the growing horizon, they introduce a significant new challenge: Both require initialization of the system for time points beyond t_0, resulting in not only a *single* IVP, requiring a single initial state $\boldsymbol{x}_0$, but many IVPs. Knowing the entire state trajectory over time is a requirement for real applications that should not be underestimated.

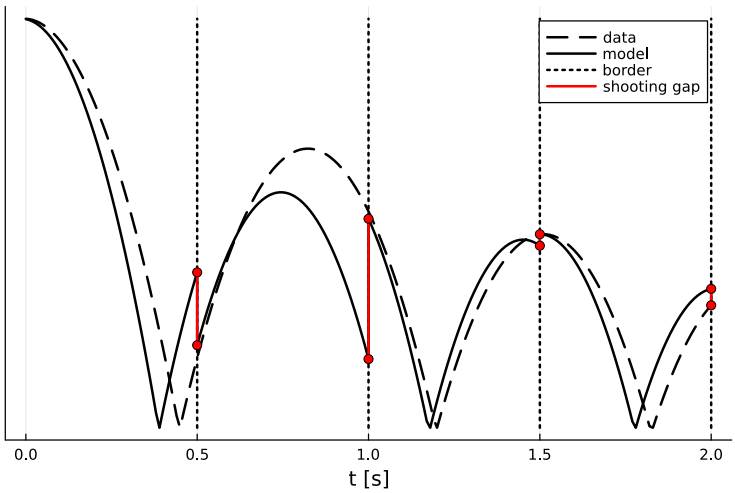


Figure 5.3: *Shooting gaps* as they occur during multiple shooting (or batching). The trajectory is *reinitialized* based on data at the shooting (or batch) borders. This may lead to large gaps (red), that later hinder the proper simulation of the entire trajectory.

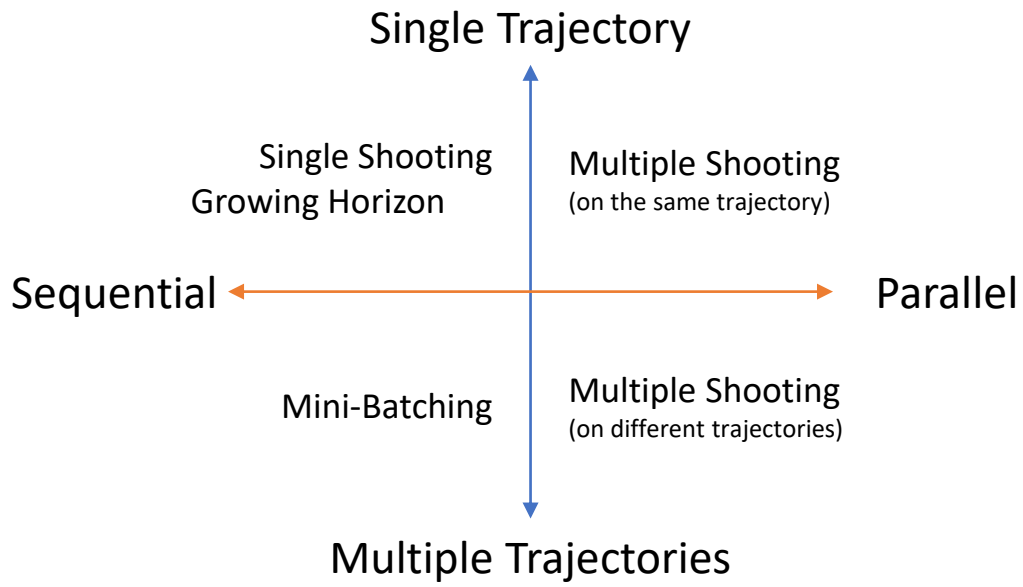


Figure 5.4: Relationship between *shooting* and *batching* methods.

5.3 Solution-based (SB)

All the introduced training techniques, single-shooting, growing horizon, batching, and multiple-shooting, have one thing in common: They operate on the solution of the ODE, which we refer to as solution-based (SB) training methods. To obtain a gradient for optimization, this requires SA over the ODE solution, as introduced in Sec. 2.1.3. Obviously, differentiation over multiple solver steps leads to long derivative chains that are costly to evaluate, compared to e.g. FFNNs that are only analyzed a single time. This chain of derivatives can be visualized in analogy to Fig. 5.5a. This is not only costly in terms of computational performance, but also memory requirements and motivates for strategies that do not require differentiation through the ODE solver.

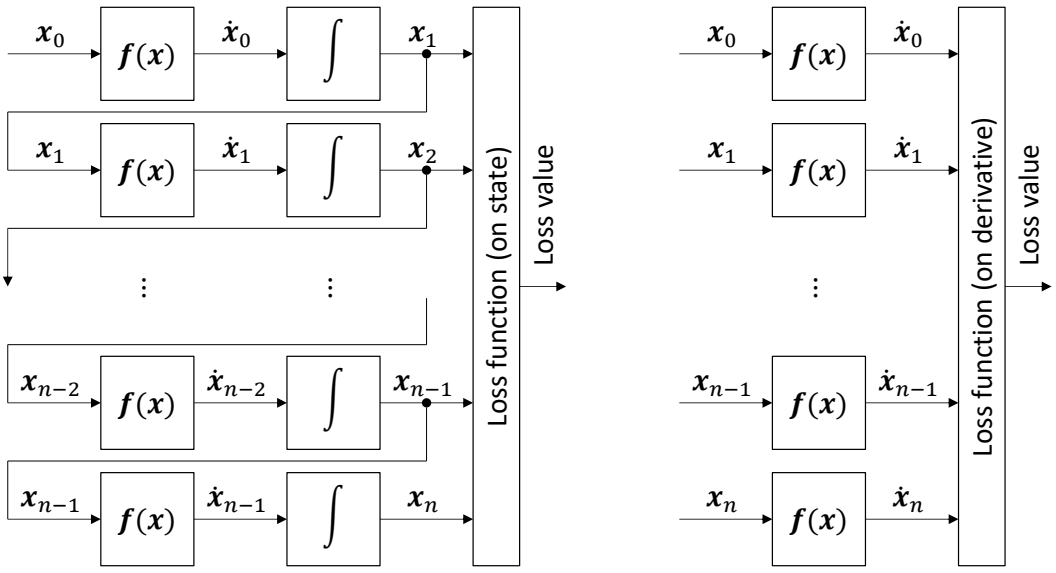
5.4 Derivative-based (DB) / Collocation

The term *collocation method* stems from solving inverse problems with DEs [99] and was established for the training of NODEs by Roesch *et al.* [100]. Interestingly, this publication already motivates the use of this strategy also for hybrid models. Collocation, which we pursue to name *derivative-based (DB) training* to prevent confusion with other terms, can be seen as method for proper initialization, even if it works differently than most other initialization methods. It involves gradient-based optimization and can therefore also be referred to as a *pre-training* step. However, as discussed during the corresponding experiment (s. Sec. 5.6.2), the local optimum found by collocation is often not sufficient, therefore the prefix *pre*, which signals that *real* training, in terms of *on the solution*, will take place afterward. The underlying idea of DB training is simple: The major source of complexity and computing time during training of UODEs comes from the fact that very long derivative chains are evaluated. This is because the propagation of sensitivities over every solver step, see Fig. 5.5a. If we cut sensitivity determination at the most complicated link of the chain — the numerical solver — the derivative chain disintegrates into multiple pieces, that are much easier (for the mathematician as well as the computer) to analyze, see Fig. 5.5b. Especially in case of implicit solvers, that require solving a nonlinear system of equations at every step, removing the solver from SA leads to significant performance gains.

However, the original cost function must be reformulated. Cost functions that depend on the system state must now depend on the state derivative (again, compare Fig. 5.5). This step (*differentiation*) from state to derivative level must not only be performed on model level (by removing the integrator), but also on the data side (cost function). Ideally, the derivatives are known from data, if not, they must be determined, e.g., by finite differences or other kernel approximations. However, the success of the DB training heavily relies on the quality of the derivative data. Even small deviations on derivative level can grow to large deviations on state level because of the accumulating behavior of the numerical integrator. This is one of the reasons why Roesch *et al.* [100] propose to apply differentiation kernels.

In theory, the solution of the DB optimization converges against the solution of the SB optimization[2], however in practice, there is often an unacceptable deviation

[2]If the constant that is lost by derivation (the initial state) is picked correctly.



(a) SB training:
The loss function is defined on the **solution** of the ODE, starting solving with $\boldsymbol{x}_0$. The remaining state trajectory is obtained by numerical integration (*solving*). The loss value is computed based on states.

(b) DB training:
The loss function is defined on collocation points of the ODE solution. The loss value is computed based on the **derivative** for multiple collocation points $\boldsymbol{x}_0, ..., \boldsymbol{x}_n$.

Figure 5.5: Comparison between training on the ODE solution and training on collocation points of a trajectory that does not need to match the ODE solution.

between both. This is because of two reasons:

Derivative approximation The derivatives (and sometimes also the states) used as reference in the optimization cost function are often only approximations of the real derivatives. Approximating the derivative too small in amount results in an integrated trajectory that is less dynamic than ground truth. However, approximating the derivatives absolutely too large can, and this is much worse, result in very stiff (or even unstable) systems. Often, too small approximations are favored therefore. As a consequence, the training objective targets a wrong integrated trajectory.

Accumulation of deviation Small deviations on derivative level can accumulate to large deviations on state level. If we consider a derivative trajectory, that only has a small offset but always in the same direction, this small error accumulates to a large deviation on state level. Even worse, the uncertainty regarding the behavior of the model increases, if the simulated state trajectory further diverges from the data trajectory. Strictly speaking, it is not known at all how the physical subsystem reacts to unknown states (beyond data) and the risk of unexpected behavior increases the more far we stray further from known data. For example in mechanics, systems may have mechanical stops, which are not approached during normal operation and are therefore not part of data.

As already stated, DB training features the big advantage of significantly im-

proved training times compared to training on the solution. Besides the deviations caused by small errors, this initialization method is not always applicable. The limitations are:

Incomplete data The state and state derivative must be known entirely. The great advantage of the SB training method is that not the entire state vector must be used in the loss function, it is sufficient (and often practical) to use only a part of the state vector to formulate a training goal, together with an initial state $\boldsymbol{x}_0$. For DB training, the entire state must be known to evaluate the system at the collocation points. Further, also the state derivative must be known from data. For large systems, this is an unrealistic requirement.

Order of differentiation By numerical integration at every step, higher order ODEs can be modeled by passing a state as derivative to the next solver step, which is quite common in engineering[3], compare Sec. 3.4.6. This naturally allows training NODEs beyond the border of integration, for example the ANN predicting the system acceleration (right-hand side) can be trained to match a given position from data. This is a remarkable fact and one reason why NODEs are so useful. At the moment the numerical solver is removed, because DB training is applied, no sensitivities can be propagated across the border of the (no longer existing) integrator. In the previous example, it is not possible to train the ANN acceleration based on a position or velocity loss function, because changing the acceleration has no influence on the position or velocity *within the same time step* — this is where we do the cut.

Of course, the second limitation is related to the first one, because if the entire state and derivative vector would be known, one could deploy a loss function that is defined on the state derivative by differentiating the original cost function (depending on the state), which would prevent the second issue. In the following, the applicability of DB training to HUDA-ODEs under real world conditions is investigated.

5.5 Solution-synchronized derivative-based (SSDB)

Because collocation or DB training can be used for parameter identification within (neural) ODEs in general, it should also be applicable for HUDA-ODEs. Whereas this technique can be applied straightforward to the function $\boldsymbol{f}$ (the ODE part of the HUDA-ODE) if the entire system state is known, this requirement is not valid for many real engineering use cases. In practice, e.g. often the discrete state is not known. For systems including events, the discrete state further changes at event instances and it is therefore required to know the discrete state over the entire simulation. In the following, a method is proposed to pre-train systems for which the system state is not known entirely.

The core idea for the derivation of a new method is to approximate the unknown part of the system state reasonably to being able to apply DB training, while avoiding differentiation over the numerical solver as in SB training to safe performance.

[3]For example, mechanical systems are usually expressed as second order ODE: Acceleration is integrated to velocity, and velocity is integrated to position.

DB training is used as starting point, but requires the complete system state for every time step which we identified a requirement which is unrealistic to satisfy. However, the current state vector can be approximated by numerical integration of the last derivative, starting with the derivative at t_0. Because some parts of the state are known from data, only the unknown part is overwritten by the previous integration step, compare Fig. 5.6. In summary, we *estimate* the unknown part of the state vector by numerical integration.

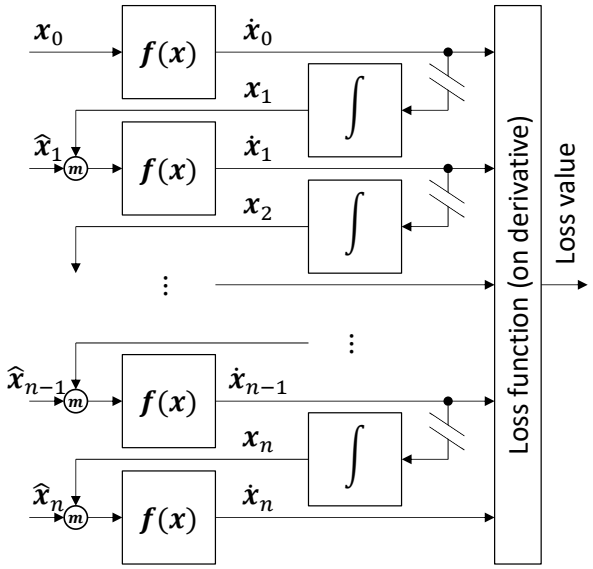


Figure 5.6: SSDB training:
The loss function is defined on the **derivative** of the ODE solution. Starting the integration with $\boldsymbol{x}_0$, the next state $\boldsymbol{x}_1$ is obtained by numerical integration. On the other hand, some entries of the state are known from data $\hat{\boldsymbol{x}}_1$. The known values from $\hat{\boldsymbol{x}}_1$ are therefore merged (by a function $\boldsymbol{m}$) into the state from the integrator $\boldsymbol{x}_1$ and solving proceeds with the merged state. Note, that no sensitivities are propagated through the solver (interrupted connection lines) and therefore no SA method for ODE solving needs to be applied, significantly reducing the computational effort compared to SB training.

We will further show, that the resulting algorithm provides performance close to DB training, while being able to process incomplete data (regarding the system state), like SB training is capable of. The resulting method can then be applied to efficiently initialize — or pre-train — hybrid models under realistic conditions (incomplete state).

As a final note and without further examination, it was observed, that it is not efficient to train DB until convergence (so no further decrease in training loss), if SB training is planned to be applied afterward. It seems like in many cases the converged solution of DB training is a suboptimal starting point for SB training. To summarize, switching from derivative- to solution-based training *before* convergence (for example after a fixed number of training steps) has often led to an improved training process in all respects.

5.6 Experiment

Within the following experiment, we compare the different introduced training types — SB, DB, and SSDB — with respect to computational performance, as well as convergence. Because application of SSDB is only meaningful if not the entire state is present within data, the experiment is performed two times. First, the entire continuous and discrete state of the bouncing ball is assumed known, and second, only the continuous part is known. In both cases, the initial state is given. To provoke a plausible problem based on the missing discrete state information, the bouncing ball is initialized as if 8 of 10 collisions had already occurred. As a result, the ball breaks after two further collisions and the dynamics drastically change starting from this point, as visualized in Fig. 5.7.

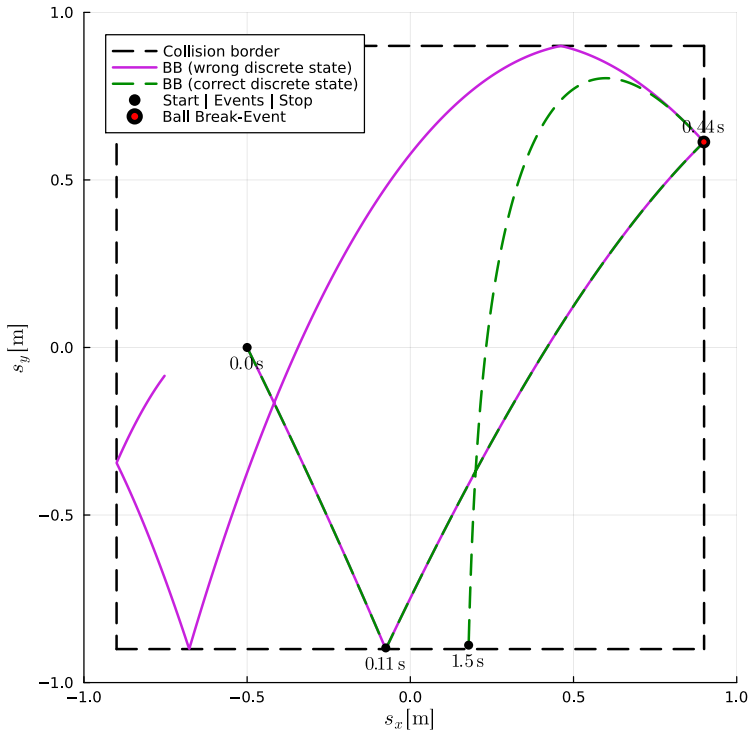


Figure 5.7: Impact of the discrete state for the bouncing ball. The bouncing ball is initialized as if $x_d = n = 8$ of $n_{max} = 10$ collisions had already occurred, which leads to the fact that the ball breaks at the marked point (red), where $n = n_{max}$. As a result, the dynamics drastically change, starting at the breaking point (broken ball in green-dashed, healthy ball in violet). Both trajectories are simulated for 1.5 s.

All experiments are performed with *growing horizon*. Details regarding the experiment can be found in Tab. 5.1.

5.6.1 Hypothesis

First, we observe that the training methods DB and SSDB should perform the same if the entire state is known, because for every time step in SSDB training, the entire state is fetched from data and no entries are corrected by the solver step. However, if the discrete state becomes unknown, DB training cannot track the change of the

Table 5.1: Setup for the training strategies experiment.

Attribute	Value
Training strategy	Growing Horizon (SB, DB, SSDB)
Horizon start	0.2 s
Horizon loss threshold	0.05
Horizon increment	0.1 s
Horizon length	1.5 s
Loss function	MAE (s_x, v_x, s_y, v_y)
Loss evaluation points	$\{0.0\,\text{s}, 0.1\,\text{s}, ..., 1.4\,\text{s}, 1.5\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam ($\eta = 0.001$)
Architecture	Sa (SOP)
ANN layout	$4 \to 8$ (tanh)
	$8 \to 8$ (tanh)
	$8 \to 2$ (identity)
Number of training steps	5,000
Number of repetitions	100

discrete state over the simulation, which inevitably leads to learning an incorrect right-hand side as soon as the ball does not break, even if it should. SSDB on the other hand will be able to keep track of the changing discrete state, because events are handled within the performed simulation steps.

In general, SB training should lead to better results, compared to DB and SSDB, because deviations on derivative level are recognized on state level and can be corrected. However, DB training is assumed to be faster in terms of wall clock time, because it requires neither performing a solver step nor differentiation over the solver. In terms of computing speed, SSDB training should perform in between SB and DB, as executing a solver step still takes place (causing some cost), but no longer involving SA (which in turn leads to significant savings).

5.6.2 Results & discussion

In the following we investigate the experiment results. First, we examine the case with known continuous *and* discrete state, and second, the case that only the continuous state is known.

Discrete state known

We start examination based on the loss function over training for the case, that the discrete and continuous state is known, see Fig. 5.8.

Typically for DB training is, that it does converge to a worse minimum compared to SB training. This is because the loss function only enforces correct *derivatives* at a finite number of points in time, and errors for derivatives are neglected entirely in between these points. However, by numerical integration, wrong derivatives can accumulate and lead to large deviations on state level. SB training, on the other hand, does enforce correct *state* values at a finite number of time points, but because

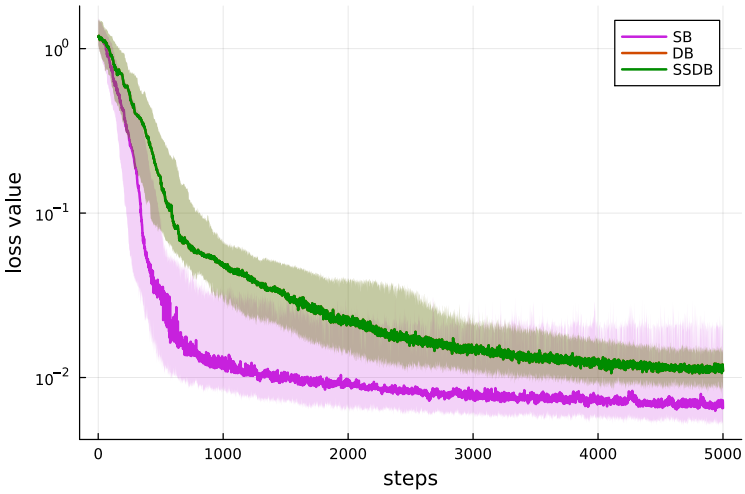


Figure 5.8: Results for the experiment where the discrete state is known. Comparison of the loss (median) for different training types. The trajectories DB and SSDB lay exactly above each other. The semi-transparent ribbons show the 25th and 75th percentiles.

wrong derivatives in between are integrated to a wrong state, this is punished by the loss function. For the chosen parameterization, DB converges to a worse minimum than SB, which is a typical observation.

The quantitative results of the experiment can be investigated in Tab. 5.2.

Table 5.2: Quantitative results for the training methods experiment (discrete state known): Median, 25th and 75th percentiles, and minimum of the loss values of all repetitions.

Experiment	Step	Median	P_{25} | P_{75}	Min
SB	500	0.029906	[0.012676, 0.084768]	0.004732
DB	500	0.15897	[0.078378, 0.298061]	0.010051
SSDB	500	0.15897	[0.078378, 0.298061]	0.010051
SB	5,000	0.006521	[0.005332, 0.021075]	0.003024
DB	5,000	0.010967	[0.008273, 0.014475]	0.003475
SSDB	5,000	0.010967	[0.008273, 0.014475]	0.003475

We further find, that SB training converges faster than DB training in terms of the number of performed training steps. However, it is very important that this is not the case for the training *duration*. In general one step of DB training is significantly computational cheaper compared to SB training, because no differentiation over the numerical solver happens. This is shown in Fig. 5.9, that visualizes the decrease in loss over the actual training *duration* in seconds.

Here, DB training is able to provide the fastest convergence rate, however then gets stuck in a local minimum and is overtaken by SB training. As in the previous figure 5.8, DB and SSDB converge to the very same minimum, even if this is visually deceptive in Fig. 5.9. As we expect, SSDB lies in between DB and SB in terms of computational time, because numerical integration is still required (in contrast to

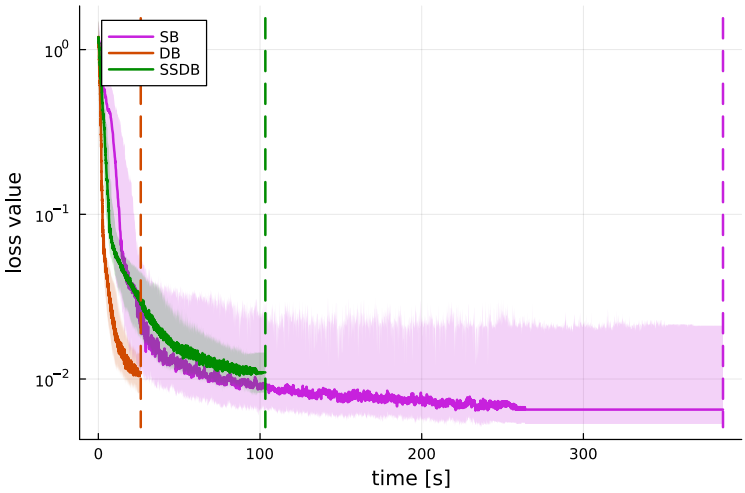


Figure 5.9: Result times for the experiment where the discrete state is known. Comparison of the loss (median) of different training types. The horizontal dashed lines show the average completion time of the experiment. The semi-transparent ribbons show the 25th and 75th percentiles.

DB training), but no differentiation over the numerical integration is performed (as for SB training).

Discrete state unknown

Further, we want to investigate the difference between DB and SSDB training if the state is *not* entirely known. As to expect, we find different progress of the median loss curve with unknown discrete state, see Fig. 5.10.

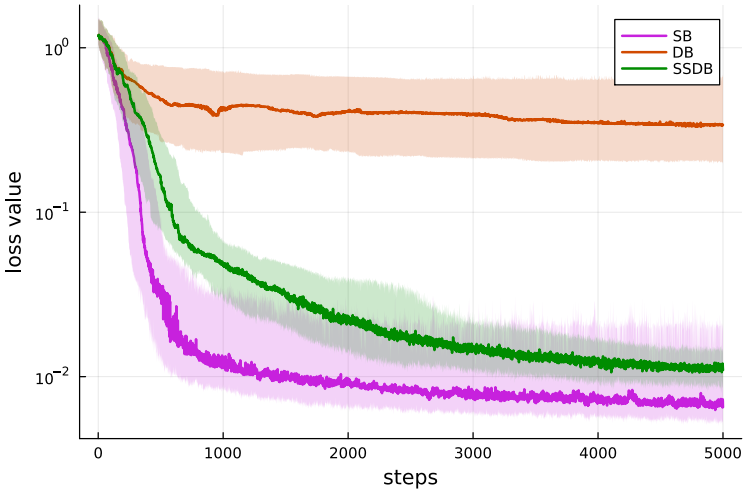


Figure 5.10: Results for the experiment where the discrete state is unknown. Comparison of the loss (median) of different training types, if only the continuous state is known. The semi-transparent ribbons show the 25th and 75th percentiles.

Whereas SB and SSDB feature the same progress of the loss curve as in the previous experiment, which is consistent with the hypothesis, the DB training is not able to learn for an acceptable representation of the system. This is because for ground-truth data, the ball *brakes* in the middle of the simulation, which means that two different right-hand sides — before and after breaking — are present. However, the ML submodel cannot access the discrete state, that stores the information of "broken", and is therefore not able to distinguish between both right-hand sides, which effectively leads to a mean value being learned. The resulting complication can be investigated quantitatively in Tab. 5.3.

Table 5.3: Quantitative results for the training methods experiment (discrete state unknown): Median, 25th and 75th percentile, and minimum of the loss values of all repetitions.

Experiment	Step	Median	P_{25} | P_{75}	Min
SB	500	0.029906	[0.012676, 0.084768]	0.004732
DB	500	0.48325	[0.268103, 0.78046]	0.046693
SSDB	500	0.15897	[0.078378, 0.298061]	0.010051
SB	5,000	0.006521	[0.005332, 0.021075]	0.003024
DB	5,000	0.339872	[0.198559, 0.665643]	0.057911
SSDB	5,000	0.010967	[0.008273, 0.014475]	0.003475

Finally, we can summarize the properties of the three approaches in Tab. 5.4.

Concerning the requirements to be applicable, SB, as well as SSDB can be interpreted as IVPs, both require the entire initial state $\boldsymbol{x}_0$ from data. DB on the other hand, additionally requires information of the entire state trajectory over time.

Table 5.4: Summary and comparison of the different training strategies.

Method	SB	DB	SSDB
Works with (partly) unknown state	✓	✗	✓
Works with (partly) unknown derivative	✓	✗	✓
Requires differentiation over solver	✓	✗	✗
Computational performance	−	+	∼
Converges to ODE solution	+	∼	∼

To conclude, DB and SSDB as only methods for training are only useful to a limited extent in the practical application. Whereas both perform very well on academic examples, they heavily rely on correct derivatives from the system to be modeled. Even for analytically known derivatives, as in the investigated example, we found that DB and SSDB converge to a suboptimal minimum. Because of the better runtime of the derivative-based approaches, they are however very interesting as pre-training or initialization routines.

Contribution 9 *Besides the presented limitations, DB training is a valuable tool, however in real life engineering it requires additional modifications to deal with (partly) unknown states and derivatives. We introduced* solution-synchronized derivative-based (SSDB) *training, that synchronizes the unknown part of the state based on the output of a simulation. We can effectively apply this method for* initialization *before the actual SB training for hybrid models, and decreases training time compared to pure SB training.*

Chapter 6

Regularization

6.1 Introduction

Regularization names a class of methods for promoting simpler, more correct, or more generic solutions to ill-conditioned problems. Especially in ML, one of the main goals is to avoid overfitting the defined optimization problem to promote better generalization [101]. Here, a distinction can be made between explicit and implicit regularization. Explicit regularization refers to the introduction of additional constraints to the cost function, which can be applied directly (by adding a term) or indirectly (by applying optimization constraints). Implicit regularization refers to other strategies that propagate a simpler solution but do not act directly as part of the optimization objective, such as data pre-processing and special training strategies. In a sense, hybrid modeling itself can be interpreted as an implicit regularization strategy because the evaluation of a physical model as part of the model definition restricts the expressiveness of the hybrid model in favor of improved training characteristics.

Open challenges
We recall some of the open challenges for hybrid modeling that we want to tackle with (explicit) regularization:

- **Stiffness & Instability:** Since the modeling of physical systems often uses equations that are derived from physical conservation laws, these are implicitly fulfilled. In multibody systems, for example, constant masses are often assumed, which implicitly fulfills mass conservation. When formulating complete equilibria of forces and torques, the conservation of momentum and energy is implicitly applied. However, for models of fluid mechanics, these three conservation equations must be formulated explicitly. A direct consequence of incorporating conservation laws is that the stability properties of physical systems are known or can be determined using conventional methods (stability analysis). Extending physical models with machine learning may destroy the validity of these conservation laws, resulting not only in loss of physical interpretability but also in the loss of explainable system properties (like stability), which has a major impact on the numerical solvability of the system. The topic of stability and stiffness is discussed in the state of the art and a new approach is investigated in Sec. 6.3.

- **Frequencies & Damping** Multiple publications [95], [102], [103] investigated, that ANNs tend to prefer learning low frequencies, resulting in neglecting high frequency parts of the system. This is a general issue when learning dynamic systems featuring multiple scales of frequencies (or time constants), like e.g. multi-domain systems. Further, for some applications, physical system properties are known, that are not necessarily represented in the measurement data. One example of this are vibrations in mechanical systems, which are not detected by the sensors due to a lack of recording rates, but can have a major influence on the system. Especially in mechanical systems, engineers often know such frequencies from dedicated experiments and material properties. Knowledge about these frequencies (and further the damping of oscillations) can be used to regularize training. As for stiffness and stability, this is further presented in Sec. 6.3.

- **Event chattering:** Hybrid models involving events tend to cause so-called *event chattering*, the unintentional, high-frequency triggering of events. This is due to the fact that assumptions made when modeling the event indicator for the physical system are no longer guaranteed by the hybrid model. Event chattering is further examined in Sec. 6.4.

- **Model assertions:** During modeling, assertions are introduced to (large) simulation models in order to ensure easier fault diagnosis, maintainability and expandability. However, these assertions are unknown from the perspective of the optimization process and are quickly violated in hybrid modeling practice. Without further precautions, this leads to the termination of the solution process (as the model can no longer be evaluated) and further to stagnation of training, as a gradient can no longer be determined. Model assertions are discussed in Sec. 6.5.

6.2 Related work

There is, of course, no state of the art for regularization of HUDA-ODEs. However, other regularization strategies from the field of modeling physical systems based on ML, can be discussed and the adaptability for HUDA-ODEs can be investigated. Some promising and representative methods are briefly introduced and discussed in the following, and new methods for explicit regularization are proposed.

Neural ordinary differential equation (NODE)
NODEs [11] can be interpreted as implicit regularization technique. The incorporation of a numerical solver improves the capability of the involved FFNN to express the solution of an ODE — technically by learning the derivative (right-hand side). However, in contrast to a pure FFNN, the neural ODE is not a UA anymore, because trajectories can easily be investigated, that cannot be expressed [104].

Physics-informed neural network (PINN)
Even though the actual focus is on replacing PDEs, the physics-informed neural network (PINN) [12] can be used to learn for an ODE solution as well. The key idea is that an ANN is applied to learn for the ODE solution, while training is

performed on the *temporal* derivative of the ANN. The loss function compares the derivative of the ANN to a given right-hand side — physical a priori knowledge — for predefined *collocation points*. The assumption is that if the derivatives at the collocation points match, also the learned ODE solution will provide a good fit. Because of extending the loss function by an additional term, comparing ANN derivative and a physical model right-hand side, PINNs can be interpreted as explicit regularization technique. In the context of regularization, the concept of PINNs can even be seen as the *explicit counterpart* (physical model in the loss function) to *implicit* hybrid modeling (physical model structurally incorporated).

Simple temporal regularization (STEER)
In Ghosh *et al.* [105], the implicit regularization strategy simple temporal regularization (STEER) is introduced. The core idea is — as the name suggests — simple: The termination time for solving the ODE is not assumed constant, but perturbed to vary between gradient determinations. In the context of stiff equations, it was exemplified that STEER can significantly improve training in terms of a smaller mean squared error (MSE) compared to training based on the unregularized loss function [105].

Hamiltonian neural networks (HNN)
Hamiltonian neural networks (HNN) learn for the Hamiltonian $\mathcal{H}$ instead of the entire right-hand side $\boldsymbol{f}$, which is the case for example in NODEs [106]. Given the Hamiltonian $\mathcal{H}$, position $\boldsymbol{q}$ and momentum $\boldsymbol{p}$ of a physical system, that fulfills energy conservation, the dynamics of the systems can be defined by

$$\begin{bmatrix} \frac{d\boldsymbol{q}}{dt} \\ \frac{d\boldsymbol{p}}{dt} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}} \\ -\frac{\partial \mathcal{H}}{\partial \boldsymbol{q}} \end{bmatrix}. \tag{6.1}$$

In analogy, a loss function fulfilling the Hamiltonian modeling approach can be derived straight forward [106]:

$$l_{HNN} = \| \frac{\partial \mathcal{H}}{\partial \boldsymbol{p}} - \frac{d\boldsymbol{q}}{dt} \| + \| \frac{d\boldsymbol{p}}{dt} + \frac{\partial \mathcal{H}}{\partial \boldsymbol{q}} \|. \tag{6.2}$$

As a consequence, training the ANN for an unknown $\mathcal{H}$ requires differentiation of the partial derivatives $\frac{\partial \mathcal{H}}{\partial \boldsymbol{p}}$ and $\frac{\partial \mathcal{H}}{\partial \boldsymbol{q}}$ w.r.t. the ANN parameters. This results in nested differentiation, which is still not supported by many AD libraries and expensive in any case. However, because of the explicit regularization via the Hamiltonian modeling approach, the resulting system full-fills energy conservation, resulting in better extrapolation and generalization [106].

Stable neural ODE for defending against adversarial attack (SODEF)
Kang *et al.* [107] propose to enforce stability (in terms of negative real eigenvalues) by deploying a special final layer to the ANN. The resulting model is referred to SODEF and focuses on classification tasks. The core idea results from two observations:

1. Points in the close neighborhood of Lyapunov-stable equilibrium positions are robust against perturbations.

2. Lyapunov-stable equilibrium positions are sometimes close to each other, which leads to small stability regions.

With the goal to separate Lyapunov-stable equilibrium points from each other and create larger stable neighborhoods this way, the cost function is extended by regularization terms (SODEF regularizers).

Stabilized NODEs
In Linot *et al.* [108], *stabilized neural ordinary differential equation* are introduced, featuring a linear and a non-linear term for the right-hand side:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{f}_n(\boldsymbol{x}), \tag{6.3}$$

where $\boldsymbol{f}_n$ is a nonlinear function and $\boldsymbol{A}$ is a parameterizable matrix, that can alternatively be replaced by a convolutional neural network (CNN). It is claimed, that the resulting models are more stable, but besides a small empirical study this is not investigated deeper. However, investigating this hypothesis mathematically could be an interesting future research task.

Quadrature adjoint
In Kim *et al.* [37], the training of stiff neural ODEs is investigated. The impact of stiff equations on the adjoint problem, that is solved for sensitivity analysis, is investigated, with the result that either the forward or backward problem leads to an unstable solution for stiff ODEs [37], [109]. To tackle the stability issues within the *classical* adjoint SA [11], [26], a new method called *Quadrature Adjoint* was developed. The Quadrature Adjoint method uses the dense output (so the interpolation polynomial) of the forward and backward solution to solve the actual adjoint sensitivity equation via *quadrature*, leading to a reduced computational cost comparing to using an ODE solver for this task [37].

Normalization
Also in Kim *et al.* [37] it is investigated, that learning a normalized neural ODE instead of the actual scale of the problem, is necessary to enable for learning stiff benchmark ODEs. To do so, the right-hand side is scaled [37]:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}(t), t) \odot \bar{\boldsymbol{x}} \cdot \frac{1}{\bar{t}}, \tag{6.4}$$

with state scaling factor $\bar{\boldsymbol{x}}$, that is given by

$$\bar{x}_i = max(x_i(t_0), ..., x_i(t_f)) - min(x_i(t_0), ..., x_i(t_f)) \quad \text{for} \quad i \in \{1, ..., |\bar{\boldsymbol{x}}|\}, \tag{6.5}$$

and time scaling factor $\bar{t} = t_f - t_0$.

To summarize, there exists a variety of strategies to deal with instability and stiffness within (neural) ODEs, and the methods introduced are only a small part. For example, Tuor *et al.* [110] investigate stability by regularizing layer weights, but only for linear and discretized NODEs and the corresponding method is therefore not further investigated in the context of highly nonlinear and industry-driven HUDA-ODEs. Further publications investigate learning for stable dynamical systems involving Lyapunov functions [111], [112], or even investigate learning stable discrete dynamical systems [113], however are not further introduced in detail, because we would like to examine a broader area here. The methods developed within this chapter are not

intended to *compete* with the methods of the presented related work w.r.t. stability and stiffness. Differently, we want to propose new methods, that open up to new applications. E.g., *eigen-informed* training — introduced in the following section — should be understood as a *more generic* approach, that goes beyond incorporation of stability and dealing with stiffness, and not as a replacement for existing methods under all conditions. A comparison (baseline) with existing approaches would shift the focus to the wrong thing and limit the possibility of demonstrating the actual area of application.

In the following, we develop different regularization techniques to tackle the open challenges.

6.3 Eigenvalues

For simulation and control engineers, working with eigenvalues is of immense value, as many properties of (linear) dynamic systems can be evaluated quickly. In the field of machine learning, there exist publications with reference to eigenvalues (see related work), but working with eigenvalues is generally considered rather exotic. However from a practical point of view, the idea of incorporating eigenvalue information into the training of hybrid models feels attractive and is discussed in this section. The section 6.3 is based largely on an own publication [114], including formulas, definitions, and observations.

6.3.1 Introduction

The eigenvalues of an ODE are defined as the eigenvalues of its system matrix $\boldsymbol{A} = {}^{\partial \dot{\boldsymbol{x}}}/_{\partial \boldsymbol{x}}$. Eigenvalues of the real valued system matrix are in general complex. For nonlinear systems, the system eigenvalues depend on the system state and therefore vary over time, and the system matrix for a given time point can be derived by *linearization*. Eigenvalue examination of nonlinear systems is feasible, if the *Hartman–Grobman Theorem* [107], [115], [116], [117] applies, that states that for equilibrium points $\boldsymbol{x}^*$ that must full-fill $\boldsymbol{f}(\boldsymbol{x}^*) = \boldsymbol{0}$, linearization is applicable if the Jacobian ${}^{\partial \boldsymbol{f}}/_{\partial \boldsymbol{x}^*}$ has no eigenvalue with a real part equal to zero. Such equilibrium points are referred to as *hyperbolic*. If applicable, the eigenvalues at equilibria (and close to the equilibria) encode many important system properties like stability, stiffness and oscillations within the dynamic system, which are investigated separately in the subsections 6.3.2.1-6.3.2.4. However, the proposed method does not exceed the common limitations of *linearization*, like for example limited significance for highly non-linear, bifurcated, or chaotic systems.

Determination of eigenvalues

Based on the linearization $\boldsymbol{A}$, the eigenvalues can be computed. Eigenvalues are defined as the solution of the equation:

$$\boldsymbol{A}\boldsymbol{v}_i = \lambda_i \boldsymbol{v}_i \quad \text{for} \quad i \in \{1, \cdots, |\boldsymbol{\lambda}|\}. \tag{6.6}$$

So for any eigenvalue λ_i, there is a corresponding eigenvector $\boldsymbol{v}_i$. Any square matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ features a maximum of n eigenvalues, so $|\boldsymbol{\lambda}| \leq n$. To find a solution, we substitute $\boldsymbol{v}$ by $\boldsymbol{I}\boldsymbol{v}$ in Eq. 6.6 and rearrange to obtain

$$(\boldsymbol{A} - \boldsymbol{I}\lambda_i)\boldsymbol{v}_i = \boldsymbol{0}, \tag{6.7}$$

which is a homogenous system of linear equations, that only has a (non-trivial) solution for $det(\boldsymbol{A} - \boldsymbol{I}\lambda_i) = 0$. This equation can be solved analytically e.g. for $n = \{1, 2\}$, however for the general case that involves larger matrix dimensions often the QR-algorithm [118], [119], [120] is applied. The algorithm can be divided into two steps: First, the QR-decomposition for the matrix $\boldsymbol{A}$ is computed, then the matrix $\boldsymbol{A}$ is updated, compare Alg. 2.

Algorithm 2 QR-algorithm for eigenvalue and -vector determination.

Input: Matrix $\boldsymbol{A}$

$i := 0$ ▷ initialize iteration counter
$\boldsymbol{A}_0 := \boldsymbol{A}$ ▷ initialize matrix with eigenvalues
$\boldsymbol{U}_0 := \boldsymbol{I}$ ▷ initialize eigenvalues
while !$converged$ **do** ▷ as long not converged
    $\Delta := ...$ ▷ compute a shift, or use zero
    $\boldsymbol{Q}_i, \boldsymbol{R}_i := qr(\boldsymbol{A}_i - \Delta \cdot \boldsymbol{I})$ ▷ QR-decomposition
    $\boldsymbol{A}_{i+1} := \boldsymbol{R}_i \boldsymbol{Q}_i + \Delta \cdot \boldsymbol{I}$ ▷ rebuild matrix $\boldsymbol{A}$
    $\boldsymbol{U}_{i+1} := \boldsymbol{U}_i \boldsymbol{Q}_i$ ▷ update matrix $\boldsymbol{U}$
    $converged := ...$ ▷ define a convergence criterion
    $i := i + 1$ ▷ increment step count
end while
Output: $\boldsymbol{A}_i$ ▷ $\boldsymbol{A}$ now holds eigenvalues on diagonal
Output: $\boldsymbol{U}_i$ ▷ $\boldsymbol{U}$ holds the eigenvectors in columns

After termination of the algorithm at step i, the eigenvalues of $\boldsymbol{A}$ can be found on the diagonal of $\boldsymbol{A}_i$, and $\boldsymbol{U}_i$ is a matrix with the corresponding eigenvectors in columns. As termination criterion for the iterative procedure, the change in the diagonal can be monitored. Multiple optimizations of the algorithm exist, for example, starting the first iteration with the Hessenberg form of $\boldsymbol{A}$, or introducing additional *shifts* Δ for faster convergence.

6.3.2 Method

The key idea of the proposed explicit regularization strategy is straightforward: First, there is a strong relation between the geometric position of eigenvalues and properties of the corresponding physical system, as deeper investigated within the following subsections. Second, if eigenvalues are computed in a differentiable way, the geometric eigenvalue positions — and system properties respectively — can be used to regularize training. The method presented in the following was published as *eigen-informed training* in Thummerer and Mikelsons [114] (preprint [121]). Concerning computational cost, we recognize that the linearization of the system $\boldsymbol{A} = {}^{\partial \dot{\boldsymbol{x}}}/_{\partial \boldsymbol{x}}$, that is required for eigenvalue determination, needs to be determined during SA on the ODE solution. Strictly speaking, this leads to *nested differentiation*, because sensitivities regarding different values need to be determined: For the ODE solution, parameter sensitivities are required, for the linearization on the other hand, we need sensitivities with respect to states.

First, the differentiable implementation of the eigenvalue computation in form of the QR-algorithm is investigated. We recall, that differentiable programming allows to differentiate the presented QR-algorithm out-of-the-box, as long as all operations

are differentiable. If we investigate Alg. 2, we find that this is the case if the sub-algorithm *qr* is implemented differentiable, which is possible without providing a proof at this point. However, for strict termination criteria, the algorithm needs to perform many steps, which results in two problems:

Cost To obtain correct sensitivities, every step of the algorithm needs to be performed on derivative level, too (AD). This is very costly, and an alternative would be advantageous.

Numerical issues Depending on the number of iterations, we found that sensitivity computation can be unstable, meaning sensitivities become `NaN`. This is because even if the QR-algorithm is numerically quite stable, the same does not need to apply to the problem on derivative level. The more iterations are performed, the higher the risk that sensitivities can not be determined.

Both problems motivate the search for a shortcut for sensitivity determination. Indeed, it is possible to derive the sensitivities over the iterative procedure independent of the number of iterations, in the event that the algorithm converges correctly. The corresponding formulas are summarized in Giles [122], for example for forward AD the sensitivities can be determined by:

$$\frac{\partial \boldsymbol{D}}{\partial \upsilon} = \boldsymbol{I} \circ (\boldsymbol{U}^{-1} \cdot \frac{\partial \boldsymbol{A}}{\partial \upsilon} \cdot \boldsymbol{U}), \tag{6.8}$$

where υ is the input variable for SA and $\boldsymbol{D} = diag(\boldsymbol{\lambda})$ is a matrix holding the eigenvalues on its diagonal. Further, the sensitivities for the eigenvectors $\boldsymbol{U}$ can be computed [122]:

$$\frac{\partial \boldsymbol{U}}{\partial \upsilon} = \boldsymbol{U} \cdot \left(\boldsymbol{F} \circ (\boldsymbol{U}^{-1} \cdot \frac{\partial \boldsymbol{A}}{\partial \upsilon} \cdot \boldsymbol{U}) \right), \tag{6.9}$$

where

$$F_{i,j} = \begin{cases} (\lambda_j - \lambda_i)^{-1} & \text{for } i \neq j \\ 0 & \text{elsewhere} \end{cases}. \tag{6.10}$$

Finally, also sensitivities for reverse AD can be derived [122]:

$$\bar{\boldsymbol{A}} = \frac{\partial \gamma}{\partial \boldsymbol{A}} = \boldsymbol{U}^{-T} \cdot (\bar{\boldsymbol{D}} + \boldsymbol{F} \circ (\boldsymbol{U}^T \cdot \bar{\boldsymbol{U}})) \cdot \boldsymbol{U}^T, \tag{6.11}$$

where γ is the output variable of SA. The corresponding sensitivities are implemented within the own package *DifferentiableEigen.jl*[1] and allow for efficient and robust SA over the eigenvalue and -vector determination, not only for the QR-algorithm.

Equipped with efficient SA over eigenvalue computation, we can investigate how differentiable eigenvalues can be incorporated into a loss function in the following.

6.3.2.1 Stability

If all eigenvalues are within the real negative half-plane, the ODE system is considered *stable*. If the largest real value of all eigenvalues is zero, the system is *border-stable*. Otherwise, if the system has at least one real positive eigenvalue, it is considered *unstable*.

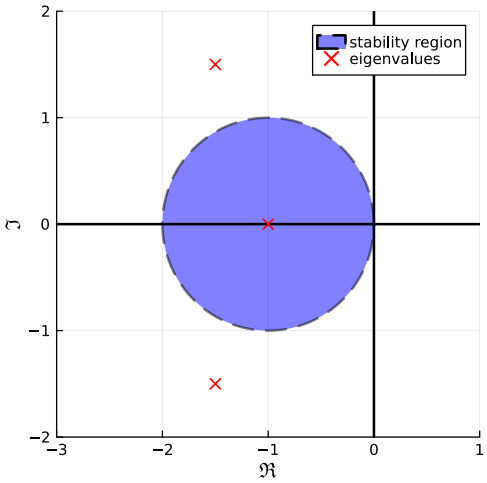
[1] `https://github.com/ThummeTo/DifferentiableEigen.jl`

Mathematically, a system is considered stable, if all eigenvalues have a real negative (and non-zero) real part, therefore $\Re(\lambda_i) < 0$ for $i \in \{1, ..., |\boldsymbol{\lambda}|\}$. A loss function, or explicit regularization term, to satisfy stability, can be defined straightforward:

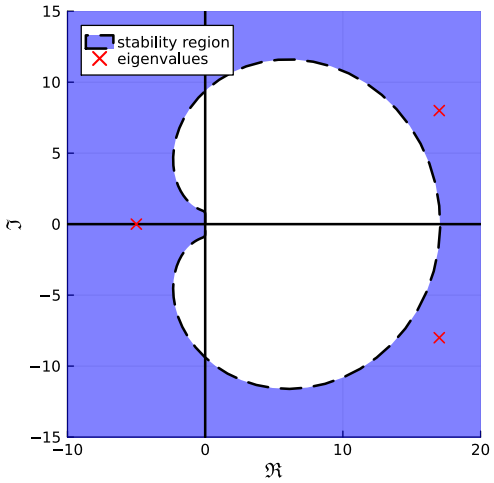
$$l_{STB}(\boldsymbol{\lambda}) = \sum_{i=1}^{|\boldsymbol{\lambda}|} \epsilon_{STB}(max(\Re(\lambda_i), 0), 0), \tag{6.12}$$

where ϵ_{STB} is a function rating the deviation between model and target, like MSE or mean absolute error (MAE). Here, the *max* function ensures, that only unstable eigenvalues with positive real part contribute to the loss value.

For the sake of completeness, whereas *stability* is a *physical* property of a system, also the *numerical* stability of the solution could be regularized, as soon as the system model is paired with a numerical solver, see Fig. 6.1. In general, satisfying numerical stability during solving is implicitly the task of the step size control algorithm within the numerical solver.



(a) An example for a physical stable (real negative eigenvalues) system, that is solved numerically unstable (eigenvalues outside stability region). The explicit Euler method is applied. Decreasing the solver step size will increase the solver stability region and allow for a numerically stable solution.



(b) An example for a physical unstable (real positive eigenvalues) system, that is solved numerically stable (eigenvalues inside stability region). The backward differentiation formula (BDF) (order 5) method is applied, which is a multi-step implicit solver.

Figure 6.1: A comparison between physical and numerical (in-)stability.

6.3.2.2 Stiffness

There are many definitions of *stiffness*, but they all agree on one point: Stiffness slows down the solution process of the (O)DE and should therefore be avoided. However, if the actual problem is stiff by design, which is common for multi-scale simulations e.g. in mechatronics, stiffness must be accepted and dealt with. The following two quotes nicely describe stiffness via the limited applicability of explicit solvers:

"Stiff equations are problems for which explicit methods don't work."

- E. Hairer and G. Wanner [123]

"Computer programs suitable for nonstiff problems do not 'blow up' in the presence of stiffness, they just become inefficient."

- L. F. Shampine and C. W. Gear [124]

To summarize, stiffness can be interpreted as the *threshold* above which it is worth switching from an explicit to an implicit method in terms of computational efficiency. In a geometric context, stiffness is related to the real range of eigenvalues, large differences indicate that slow and fast processes are part of the simulation model and therefore the fast parts dominate the step size of the solver. This range can be embedded into a loss function as follows:

$$l_{STF}(\boldsymbol{\lambda}) = \epsilon_{STF}\left(\frac{max(\Re(\boldsymbol{\lambda}))}{min(\Re(\boldsymbol{\lambda}))}, \tilde{\sigma}(t)\right), \tag{6.13}$$

where $\tilde{\sigma}(t)$ is the target stiffness and ϵ_{STF} is again an error function. However, this error function can further allow for a *corridor* around $\tilde{\sigma}(t)$, meaning it does not need to enforce an *exact* stiffness but allows for deviations within a given range. Regularizing for $\tilde{\sigma}(t) = 0$ will promote a system that is as less stiff as possible, leading to a fast simulation model [114].

6.3.2.3 Time constant

In analogy, the *time constant* of an eigenvalue is highly related to stability and stiffness, because it only depends on its real position. The time constant is defined as the negative inverse of the eigenvalue real part:

$$\tau(\lambda_i) = \frac{-1}{\Re(\lambda_i)}. \tag{6.14}$$

A straightforward regularization strategy rates the time constant $\tau(\lambda_i)$ within a loss function, however, this leads to solvability problems for $\Re(\lambda_i) \approx 0$, so for eigenvalues close (or on) the imaginary axis. Further, real eigenvalues close to the imaginary axis are penalized excessively, which is not intended in general. Therefore, a loss function to penalize the inverse — so actually the real part of the eigenvalue — is proposed:

$$l_{TIC}(\boldsymbol{\lambda}) = \sum_{i=1}^{|\boldsymbol{\lambda}|} \epsilon_{TIC}(\Re(\lambda_i), \frac{1}{\tilde{\tau}_i(t)}), \tag{6.15}$$

where $\tilde{\boldsymbol{\tau}}$ are the target time constants. In practice, the target time constants will be not known exactly, and the error function ϵ_{TIC} will be designed to allow for a range of time constants.

In the following, we examine properties which also depend on the imaginary part of eigenvalues.

6.3.2.4 Oscillation

Oscillation is a phenomenon that can be observed in physical systems of different domains. It is characterized by the repeating movement of an object or particle between two extrema [125]. In the eigenvalue representation, oscillations can easily be recognized by so-called *complex conjugate eigenvalue pairs*, so pairs of eigenvalues that share the same real part, but negated imaginary parts. Like for non-paired eigenvalues, conjugate complex eigenvalue pairs on the left side of the complex plane refer to a stable, therefore a *fading* oscillation, whereas eigenvalue pairs on the right side correspond to unstable oscillations, which *swing up*.

Assuming that an optimizer only does *small* steps and therefore the eigenvalue positions within an optimization step only move *a little*, we need to force eigenvalues into the same real part, to allow for a separation of the imaginary parts. If the real parts are close enough, the corresponding eigenvalues can form a conjugate complex pair with non-zero imaginary parts. We can define a suitable loss function for the synchronization of the real parts of an eigenvalue pair $\{\lambda_a, \lambda_b\} \in \mathbf{\Omega}$, where $\mathbf{\Omega}$ corresponds to a set of eigenvalue pairs, as follows:

$$l_{OSC}(\mathbf{\Omega}) = \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{OSC}(\Re(\lambda_a), \Re(\lambda_b)), \tag{6.16}$$

where ϵ_{OSC} is a function rating the deviation between the two eigenvalue real parts.

Frequency
Besides the oscillation capability itself, further details of the oscillation can be investigated based on eigenvalue positions. For example, the frequency f in Hz of the oscillation depends on the absolute imaginary part of the pair:

$$f(\lambda_i) = \frac{|\Im(\lambda_i)|}{2 \cdot \pi}. \tag{6.17}$$

For a given target frequency $\tilde{f}_{ab}$ for an eigenvalue pair $\{\lambda_a, \lambda_b\} \in \mathbf{\Omega}$, we can define the explicit regularization term for the frequency:

$$l_{FRQ}(\mathbf{\Omega}) = \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_a)) + \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_b)). \tag{6.18}$$

This directly leads to a frequency of 0 Hz for non-oscillating eigenvalues with zero imaginary part — just as we expect. Because the frequency is the same for both eigenvalues of a pair, we can further simplify the expression to only depend on one eigenvalue λ_a or λ_b of the pair:

$$l_{FRQ}(\mathbf{\Omega}) = 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_a)) \tag{6.19}$$

$$= 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\mathbf{\Omega}} \epsilon_{FRQ}(\tilde{f}_{ab}, f(\lambda_b)). \tag{6.20}$$

In theory, this would allow to only compute sensitivities for a single eigenvalue per eigenvalue pair, however this step requires additional modifications to the introduced QR-algorithm and corresponding differentiation rules.

Damping
Similarly, the damping $\delta(\lambda_i)$ of an oscillation can be derived from the angle between the imaginary parts and the origin:

$$\delta(\lambda_i) = \frac{-\Re(\lambda_i)}{|\lambda_i|} = \frac{-\Re(\lambda_i)}{\sqrt{\Re(\lambda_i)^2 + \Im(\lambda_i)^2}}. \tag{6.21}$$

In analogy to the regularization term for the frequency, we can define the function that regularizes the damping of an oscillation to match a target damping $\tilde{\delta}_{ab}$:

$$l_{DMP}(\boldsymbol{\Omega}) = \sum_{(\lambda_a,\lambda_b)\in\boldsymbol{\Omega}} \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_a)) + \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_b)). \tag{6.22}$$

Again, as for the frequency, damping for both eigenvalues within a conjugate complex eigenvalue pair is the same, so we can further simplify the expression to depend only on one of the eigenvalues:

$$\begin{aligned} l_{DMP}(\boldsymbol{\Omega}) &= 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\boldsymbol{\Omega}} \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_a)) && (6.23)\\ &= 2 \cdot \sum_{(\lambda_a,\lambda_b)\in\boldsymbol{\Omega}} \epsilon_{DMP}(\tilde{\delta}_{ab}, \delta(\lambda_b)). && (6.24) \end{aligned}$$

Besides the ones introduced, there are further system attributes that relate to the system eigenvalues, e.g. the natural frequency or (approximated) settling time. Of course, such properties can be used for eigen-informed training, too. To summarize, different system properties can be enforced by explicit regularization based on differentiable eigenvalue positions. It is important to mention, that especially the error functions ϵ_ϕ with $\phi \in \{STF, FRQ, DMP\}$ can be used to define geometric ranges instead of enforcing specific values for stiffness, frequency and damping.

Because eigen-informed training requires differentiation of the linearization w.r.t. parameters, computation of the parameter gradient introduces the need for *nested AD*. This is investigated in detail in Sec. A.5. The Jacobians for the involved eigenvalue calculation are determined by the differentiation rules Eq. 6.9 and 6.11, which are expensive[2] because both involve matrix inversion of $\boldsymbol{U}$. The linearization $\boldsymbol{A}$ can be shared with other numerical algorithms, like e.g. implicit solvers, which leads to computational benefits. A brief empirical study on the computational performance of eigen-informed training is part of Thummerer and Mikelsons [114].

We investigate eigen-informed training in Sec. 6.7.1 in an example system.

6.4 Events

Events are an artificial measure to allow for discontinuous changes in the simulation and require a more complex simulation process. The additional effort is rewarded by the fact that discontinuities can be represented as such and do not have to be approximated by very steep continuous functions, which require being solved with very small steps. However, events can also lead to unintended behavior, especially as

[2]The cost for matrix inversion is $\mathcal{O}(n^3)$.

soon as they are paired with ML models in hybrid modeling. This boils down to the fact that events are unintentionally triggered, because the underlying assumptions (*indicators*) are not valid within the hybrid model anymore. For such cases, this section handles the explicit regularization of events.

6.4.1 Introduction

Events are indicated by event indicators, that are designed during the modeling of the system, or could be learned by a ML model like in Chen *et al.* [16]. For state events, these event indicators directly depend on the system state, or on variables that again depend on the system state. *Event chattering* describes the unintentional, high-frequency triggering of events, often caused by a combination of several event indicators. High quality physical models should not exhibit event chattering, as this has already been taken into account and prevented during modeling. However, at the moment the system dynamics are manipulated (by hybrid modeling), event chattering might become an issue, because assumptions regarding event indicators from the time of model development may no longer be valid. At the example of the bouncing ball, it is assumed that the ball switches the velocity direction right after event handling, which does not need to be satisfied for a hybrid model manipulating the velocity[3].

It is important to state, that event chattering is not only computational expensive and undesirable, but can lead to termination of the simulation and therefore termination of the overall training process because of numerical issues, like failed detection of an event indicator root. In such cases, the proposed method is not intended to only *optimize* the training process, like e.g. eigen-informed regularization, but to *make training possible* in the first place.

6.4.2 Method

In principle, there are two possible points of intervention to regularize events: The event indicators, which are used to *localize* events, and the discrete state, which can be interpreted as *consequence* of the processed event. We observe that the discrete state, that changes discontinuously at event instances, is challenging to use within a regularization term, because its choice heavily depends on the individual model. For example, it is not required that the discrete state actually changes at an event instant. Event indicators, on the other hand, are much more suitable because they are (if modeled properly) continuously differentiable — at least in close distance to the actual zero crossing. In the following, we investigate the possibility of regularizing event indicators in order to avoid (specific) events.

Avoiding events

The first idea for regularization is to simply avoid specific events in general by penalizing the corresponding event indicators. Because events happen exactly when the event condition function $\boldsymbol{c}$ shows a zero crossing at the corresponding entry,

[3]Note, that for example an ANN can be parameterized so that the velocity before impact (negative sign), as well as the velocity after impact (positive sign), are mapped on negative values — resulting in the bouncing ball tunneling the floor.

events can be prohibited by regularization with an additional loss function term ***EV****ent* ***A****void*:

$$l_{EVA}(\bar{\boldsymbol{z}}) = \frac{1}{n \cdot |\bar{\boldsymbol{z}}|} \sum_{i=1}^{n} \sum_{j=1}^{|\bar{\boldsymbol{z}}|} \frac{1}{\|\bar{z}_j(t_i)\|_\epsilon \cdot \beta_j + \alpha_j}, \tag{6.25}$$

where $\bar{\boldsymbol{z}}$ is a vector with indices of event indicators to be *avoided*, and $\bar{z}_j(t_i)$ the j-th element of $\bar{\boldsymbol{z}}$ for time t_i. Further, n corresponds to the number of solver steps, or saved steps respectively. The hyperparameter $\boldsymbol{\alpha} > \mathbf{0}$ controls the maximum penalties (which are $1 \oslash \boldsymbol{\alpha}$), $\boldsymbol{\beta} > \mathbf{0}$ is the nonlinearity of regularization (the larger, the stronger), and ϵ is an error norm (like e.g. L1 or L2).

The major downside of this approach is that we already punish operating the system *close* to an event, which is not intended. We only want to regularize events that are *going to* happen, but not events that *might* happen. The fundamental observation is that being close to an event is not an issue, as long as we do not continue to move in direction of the event. Note, that an event indicator value can be very small (which is rated *close to an event*), however, its derivative can indicate that we are actually moving away from the event. Nevertheless, a regularization term as in Eq. 6.25 will penalize this. In practice, models are often operated *close* to events, for example an industrial robot with mechanical stops (modeled by events) will operate close to events without triggering them during regular operation — well-functioning industrial robots should not exceed their mechanical limits. Such circumstances are common in modeling and make it difficult to perform regularization by evaluating event indicators, and to find meaningful parameters for the intensity of this measure.

Especially at our recurring example — the bouncing ball — we can prove by a simple derivation that the method as proposed is not applicable. A common optimization objective for the considered example system is to match a given ball position. Note, that the event indicators almost equal the ball position, only small offsets for the wall positions are added. We now apply the secondary objective (regularization term), which is to not trigger the event indicator, as introduced. Because the event indicator for our bouncing ball almost exactly matches the ball position — the same as the primary objective — we have opposite optimization directions defined in the primary and secondary objectives, which corresponds to a badly conditioned optimization problem.

Regularizing the event frequency

Based on these observations, we can develop a more effective way of thinking: We do not want to penalize proximity to an event, but only if we actually *trigger* an event. From a user perspective, to obtain a simple quantity for regularization, one could use the *average event frequency* of the simulation:

$$\Delta t_{eve} = \frac{|\boldsymbol{E}|}{t_f - t_0}, \tag{6.26}$$

where $|\boldsymbol{E}|$ is the number of events. However, there are two major restrictions coming with that idea. First, the differentiation of the operation $|\boldsymbol{E}|$ with respect to parameters is not straightforward. Second, the average event frequency is in general not sensitive w.r.t. small perturbations, meaning small changes in parameterization do not necessarily influence the average event frequency. At the example of the bouncing ball, consider that small parameter variations indeed influence the position of

the events, but not necessarily lead to an increasing or decreasing *number* of events — and therefore an increasing or decreasing average event frequency Δt_{eve}.

Nevertheless, the core idea is appealing, and we are making another attempt by introducing a related quantity, the *estimated* average event frequency. The main difference is that we no longer focus on event locations, where the event indicator is 0, but rather on the points (or saving points) of the solution. For any of such points, we can assess whether we are approaching or moving away from an event. This no longer corresponds to a static view (*distance to the event*), but a dynamic one (*velocity at which we approach an event*). Further, if we know the distance, as well as the velocity we are approaching an event, we can make a *prediction* on when the event is going to happen — we can approximate the event distance w.r.t. the independent variable: the time. In summary, we want to estimate the time it takes for a given solution point (state and time) for the next event to happen.

In practice, *making a prediction* based on a rate of change (derivative) boils down to a *differential equation* that is solved by *numerical integration*. Indeed, we can solve the event indicator at the next time instant by applying a numerical integration scheme, like the explicit Euler method:

$$\boldsymbol{z}(t+h) = \boldsymbol{z}(t) + \frac{d\boldsymbol{z}(t)}{dt} \cdot h, \tag{6.27}$$

with step size h and event indicators $\boldsymbol{z}$. To allow for individual step sizes for every event indicator — which is quite uncommon for numerical integration but necessary for the further derivation — we can apply a vectorial step size $\boldsymbol{h}$ as in:

$$z_i(t+h_i) = z_i(t) + \frac{dz_i(t)}{dt} \cdot h_i \quad \text{for } i \in \{1, ..., |\boldsymbol{z}|\}. \tag{6.28}$$

Because we want to predict the individual event locations $t + h_i$, where all event indicators z_i are zero, we can simplify:

$$\boldsymbol{0} = \boldsymbol{z}(t) + \frac{d\boldsymbol{z}(t)}{dt} \cdot \boldsymbol{h}. \tag{6.29}$$

Now, we solve for the individual step sizes of the integration to reach the event points:

$$\boldsymbol{h} = -\boldsymbol{z}(t) \oslash \frac{d\boldsymbol{z}(t)}{dt}. \tag{6.30}$$

As a result, the vector $\boldsymbol{h}$ holds the *estimated time* until the next event location, separated for every event indicator. Of course, more advanced integration schemes could be applied as well, in order to enhance the prediction, or multiple steps could be performed. However, the explicit Euler scheme excels with very low computational cost and is sufficient for a rough estimate here. Note, that $\boldsymbol{h}$ may contain negative values as well, meaning the intersection (event) is determined to lay in the past.

The estimation for the next event time can be performed for every state on the trajectory and can be summarized in the *estimated average event frequency*:

$$l_{EFR}(\boldsymbol{z}) = \frac{1}{|\boldsymbol{z}| \cdot n} \sum_{i=1}^{n} \sum_{j=1}^{|\boldsymbol{z}|} \frac{1}{max(\epsilon, \|h_j(z_j(t_i), t_i)\|)}, \tag{6.31}$$

for $\boldsymbol{z}$ the event indicators. A small value for $\epsilon > 0$ prevents division by zero. Note, that we applied the inverse of the estimated event delta time h_j to obtain the corresponding event frequency.

Concerning limitations, we need to note that the presented procedure corresponds to identification of the next event time by performing a single integration step. Of course, this is only a linear approximation, and depending on the complexity of the event indicator function and actual distance to an event, the identified estimated event time will contain an error of varying severity. Nevertheless, we will see that the method presented can reliably help to regularize the event frequency based on this rough estimate.

Favoring events

In analogy to preventing events by regularization of the event indicators, in the rare case that the exact event locations are known, we can *favor* events. Event locations can be formalized by a tuple $\boldsymbol{e} = \{t^*, i^*\}$, where t^* is the event time and i^* is the event index, that triggered the event. We can then collect the event locations $\boldsymbol{E} = \{\boldsymbol{e}_1, ..., \boldsymbol{e}_m\}$ and define a loss function term $\boldsymbol{EVent\ Favor}$:

$$l_{EVF}(\boldsymbol{E}, \boldsymbol{z}) = \sum_{\{t^*, i^*\} \in \boldsymbol{E}} \|z_{i^*}(t^*)\|_\epsilon, \tag{6.32}$$

where $z_{i^*}(t^*)$ is the event indicator value for index i^* at time t^*. Strictly speaking, this regularization term enforces the event indicator to be zero for the known events in $\boldsymbol{E}$. However, this requires knowledge of which event indicator fires at which time. In general, one could also think about regularizing events to happen at specific points in state space. However, both cases require extracting this knowledge from data, or assuming that this is knowledge of the modeler. This requirement is unrealistic for non-academic applications and is therefore not deeper investigated, but mentioned for the sake of completeness.

We summarize our observations in the corresponding experiment in Sec. 6.7.2.

6.5 Errors

Similar to the regularization based on events, preventing error instances can be used to improve training, too. This is investigated in more detail within this section. As a side note, an attempt is being made to standardize the general concept of *error indicators* as part of a FMI layered standard for SA of FMUs in Thummerer *et al.* [126].

6.5.1 Introduction

While working with various industrial simulation models, a recurring problem occurs for hybrid models: By manipulation of the original system of equations, some implicit assumptions are violated (for example conservation laws or parameter boundaries). For large simulation models, these assumptions are often secured by implementing *assertions*[4], that throw a model error when violated. This in general results in the termination of the simulation, which cancels the training procedure.

[4] *Assertions* in terms of code assertions in software, in Julia e.g. by the command `@assert`.

6.5.2 Method

The core problem here is not, that assertions are implemented — indeed this is a good thing to keep large models maintainable and easy to debug — but that assertions are implemented *invisible* for the training process or the optimization algorithm respectively. The optimizer is not aware of model errors, that result from its parameter adaptions. One strategy to deal with this is to provide information about approaching errors *before* they happen. This can be accomplished by implementing a distance measure, further referred to as *error indicator*.

Even if assertions are defined as binary event (assertions are *true* during regular operation and *false* if an error happens), they can often be reformulated as continuous functions. For example, assertions are often used to limit values, in this case a straightforward error indicator would be the distance to this limit. One major difference compared to event indicators is that the sign of an error indicator matters, whereas it has no meaning for event indicators. In the following we define that negative values for an error indicator indicate the *distance* to the corresponding error, a value of 0 indicates the error, and positive values indicate how advanced, deep or significant the error is. Further, if error indicators $\boldsymbol{\delta}$ are available *differentiable*, errors can actively be prevented by regularization:

$$l_{ERR}(\boldsymbol{\delta}) = \sum_{i=1}^{|\boldsymbol{\delta}|} \alpha_i \cdot max(\delta_i - \beta_i, 0), \tag{6.33}$$

where again $\boldsymbol{\alpha} \geq \mathbf{0}$ are hyperparameters to control the intensity of regularization. Further, the maximum is formed with the error limits $\boldsymbol{\beta}$. If the error limits are chosen to be zero $\boldsymbol{\beta} = \mathbf{0}$, errors can be detected and recovered. However, if there is sufficient expert knowledge regarding the model, the error limits can be chosen negative $\boldsymbol{\beta} < \mathbf{0}$ to apply security margins that hit *before* the error is triggered.

It is important to note, that even if the concepts of event and error indicators appear very similar at first glance, there is a significant difference. While we allow to *dive* into errors — meaning we can detect errors after they happened by checking the error indicator and sign — this is in general not possible for events, because the event indicator is not required to change sign[5] after event handling is performed. This fact allows us to apply the *max* function for error indicators, that allows to only regularize errors once they have happened, and explicitly not if we are only *close* to an error. These observations are further investigated in the corresponding experiment, see Sec. 6.7.3.

6.6 *LimIter*: Limiting the iteration count

During working on various use cases (compare Cha. 8), an interesting observation was made, that leads to proposing another method for implicit regularization of hybrid models, or NODEs in general.

If mini-batching is applied, the computation time of different batch elements can vary by multiple orders of magnitude. This is because of varying stiffness between different batch elements, which again leads to a varying number of solver steps.

[5]For example, if monitoring the event indicators of the bouncing ball — the ball position — during regular simulation, they *never* switch sign at event instances.

Especially if explicit methods are applied (e.g. in Sec. 8.2 and Sec. 8.3), and stiff or close-to-instable regions of the optimization landscape are investigated, the training process often slows down to the point where stagnation cannot be excluded and abortion is required.

The important observation here is that for a given parameterization often only *some* batch elements exhibit this problem, while other elements can be solved with reasonable step size, so a gradient can be computed, and the parameters can be updated — maybe resolving the issue of the unsolvable batch element. This observation can be used to derive a simple strategy: If an element is difficult to solve, then stop and try another one. From a technical perspective, such an behavior can be implemented easily by specifying[6] a maximum number of iterations for the numerical solver. If the solution process is terminated after the fixed number of steps, there are two possibilities to proceed:

(a) We discard the solving attempt and request a new batch element from the batch scheduler, or ...

(b) ... we obtain a partial solution, containing only the first points of the solution until we reached the maximum number of solver steps, and calculate a loss on this partial solution. This allows to further use the already processed information, and additionally — because of the reduced time span — a positive regularization effect similar to STEER. After applying the optimization step, we proceed requesting a new batch element from the batch scheduler.

We proceed calling this strategy *LimIter* for "limiting iterations". For *LimIter* we skip providing a dedicated experiment, but apply it for all validation use cases in Cha. 8. Without providing proof, we claim that applying this technique reduced the wall-clock time for hyperparameter optimization by roughly one order of magnitude — and was therefore applied to make the hyperparameter optimization even feasible in reasonable time on a regular desktop computer.

6.7 Experiments

In this experimental section, we want to investigate different regularization strategies based on eigenvalues, as well as event and error indicators.

6.7.1 Eigenvalues

In the following experiment we want to show, that incorporation of eigenvalue information leads to better generalization of the learned hybrid model. Further benefits of eigen-informed training, like significantly improved convergence, learning on undersampled data, and even reducing the number of solver steps, is discussed in detail in Thummerer and Mikelsons [114].

As example system we use the bouncing ball again. To showcase the benefits of eigen-informed training, the system can be slightly modified: The bouncing ball is extended by a linear spring, that connects the moving bouncing ball to the center of the coordinate frame. This allows the bouncing ball to *oscillate* with a fixed

[6] In *DifferentialEquations.jl* straightforward the keyword `maxiters` to the `solve`-call.

frequency (given by the spring stiffness) and allows drawing new kinds of figures, compare Fig. 6.2.

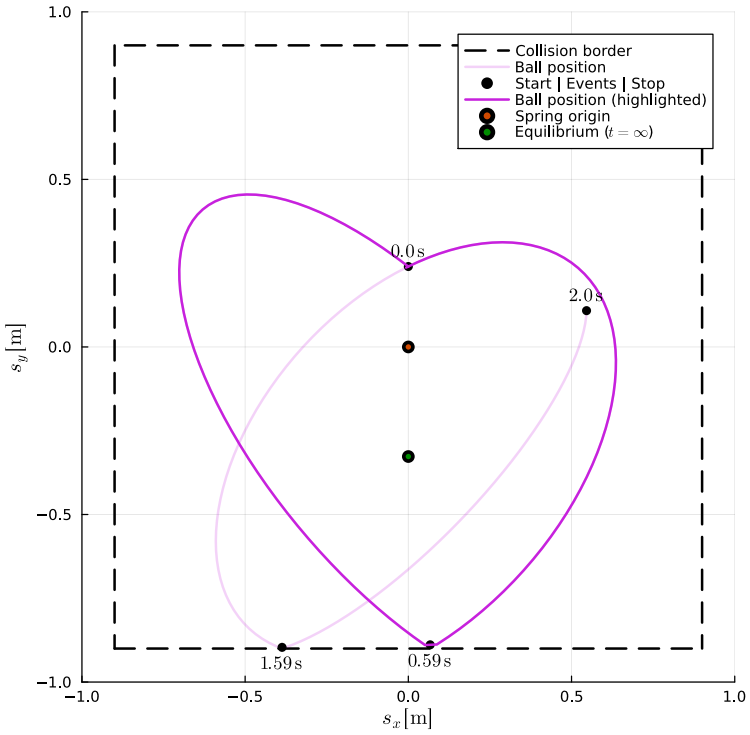


Figure 6.2: By attaching a spring to the bouncing ball, we can express our *love* for SciML.

If we want to extend the bouncing ball by a spring — switching to the perspective of an engineer in the filed — there are multiple ways of doing this:

- We assume the spring to be *known*, so there are no real uncertainties in terms of dynamics and parameters. We extend the right-hand side by a linear spring equation with fixed parameters.

- We are sure, that the spring model is a reasonable choice, but we have concerns that the parameterization might be incorrect. In this case, we can also extend the right-hand side, but keep the parameters trainable.

- We have a *guess* on the final model properties, but are not sure if the spring is actually a linear spring, or if there is, e.g., some minor nonlinearity in it. In this case, we can *learn* for a spring model, but *guide* a correct solution by explicit regularization using eigenvalues. Exactly this is done within this experiment.

- For the sake of completeness, a fourth option is regularization by evaluation of the linear spring model, as in *physics-informed* approaches [12]. This is not further investigated at this point.

The experimental setup is summarized in Tab. 6.1.

Table 6.1: Setup for the eigen-informed regularization experiment.

Attribute	Value
Training strategy	Growing Horizon (SB)
Horizon start	0.1 s
Horizon loss threshold	0.01
Horizon increment	0.1 s
Horizon length	2.0 s
Loss function	MAE (s_x, s_y) + $l_{FRQ}(\alpha = 0.001)$ + $l_{DMP}(\alpha = 0.01)$ + $l_{STB}(\alpha = 0.001)$ + $l_{OSC}(\alpha = 0.01)$
Loss evaluation points	$\{0.0\,\text{s}, 0.1\,\text{s}, ..., 1.9\,\text{s}, 2.0\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam w. exp. decay ($\eta_{start} = 0.01$, $\eta_{stop} = 0.001$, $\lambda = 0.0005$)
Architecture	PSa (SOP)
ANN layout	$6 \rightarrow 8$ (tanh) $8 \rightarrow 8$ (tanh) $8 \rightarrow 2$ (identity)
Number of training steps	7,500
Number of repetitions	30

System properties

The core idea of eigen-informed training is to reduce the amount of training data by incorporating knowledge of system properties that can be expressed by geometric properties of the eigenvalues of the system. In many applications, a priori knowledge is available for the system to be modelled. At the example of the bouncing ball with spring, an engineer in the field might examine the following system properties:

- The system is stable, so perturbations of the ball are cancelled out over time and the system comes to rest (energy dissipation). The system further features only a single equilibrium position, that is reached for $t \rightarrow \infty$ for every possible start position independent of the initial velocity. Further, there is no instable equilibrium for a spring with zero relaxed length.
- We assume a linear (or almost linear) spring, and properties like the spring stiffness might be available from a data sheet or can be determined by a very easy experiment (attach known mass, measure length increment).
- Further, the quadratic speed dependent damping factor can be approximated by a similar simple experiment (measure the time of the falling ball over a known range and fit the drag coefficient).

The eigenvalues of the ground truth system are shown in Fig. 6.3. However, it is important to keep in mind that these are (in general) not available in a real application and are only shown for information purpose.

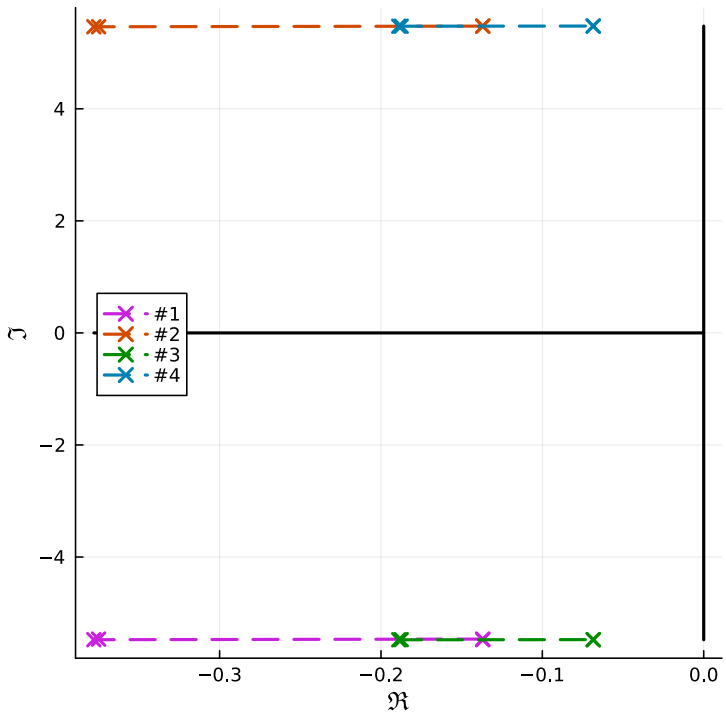


Figure 6.3: The four eigenvalues of the bouncing ball over time.

Investigating the eigenvalues, we can apply knowledge from systems and control theory and find, that the bouncing ball ...

- ... is a nonlinear system (eigenvalues change over time),
- ... is a stable system (all eigenvalues on the left half-plane),
- ... is an oscillating system (two conjugated complex eigenvalue pairs),
- ... is a damped system (all eigenvalues on the left plane and none on the imaginary axis),
- ... the damping changes over time (distance to complex origin), but
- ... the frequency is constant over time (constant distance to real axis).

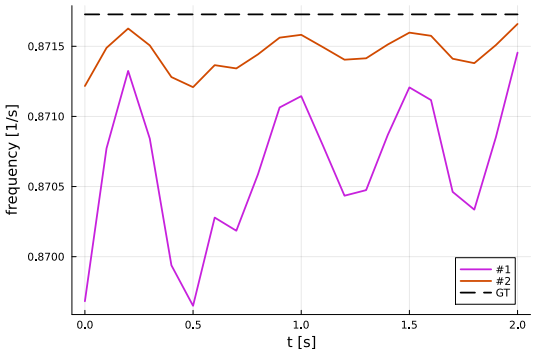
If we assume to know parameters such as the spring constant $c = 30\,\mathrm{N/m}$ and the (quadratic) damping constant $\mu = 0.075\,\mathrm{kg/m}$, and are able to extract the maximum velocity of the ball $v_{max} \approx 5.01\,\mathrm{m/s}$ from data, we can determine a target frequency

$$f_{max} = f_{min} = \frac{\sqrt{\frac{c}{m}}}{2\pi} = 0.871728\frac{1}{s} \tag{6.34}$$

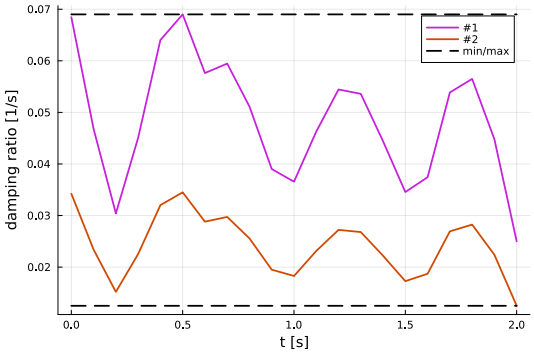
and an upper bound for the damping

$$\delta_{max} = \frac{v_{max} \cdot \mu}{\sqrt{c \cdot m}} \approx 0.068621\frac{1}{s}. \tag{6.35}$$

Based on the target frequency and maximum damping, we can apply the proposed loss function extensions introduced, aiming on a specific frequency (s. Fig. 6.4a) or limiting the maximum damping (s. Fig. 6.4b).



(a) The frequencies for the two modes, as well as the target frequency. All trajectories are quantitatively very close to each other (compare the scale of the frequency axis), and the deviation is caused by numerics.



(b) Damping values of the two modes, together with the damping limits.

Figure 6.4: Frequency and damping (and boundaries) for the bouncing ball with spring.

In the following, we train for the nonlinear friction by air resistance and additionally extend the training objective by eigenvalue information:

SOL As baseline to compare with we use the hybrid model training without eigenvalue regularization.

SOL+FRQ+OSC We apply regularization based on the knowledge that the system oscillates (eigenvalue pairs) at a known frequency.

SOL+DMP+OSC We apply regularization based on the knowledge that the system oscillates (eigenvalue pairs) and features a maximum damping.

SOL+STB We apply regularization based on the knowledge that the system is stable (negative real parts).

6.7.1.1 Hypothesis

Because of the very limited amount of data (single trajectory), we expect only limited extrapolation capability from the hybrid model. To investigate the generalizability, we introduce a further, very different trajectory for the bouncing ball, see Fig. 6.5. For the eigen-informed hybrid model, we expect better extrapolation on the unknown trajectory in particular.

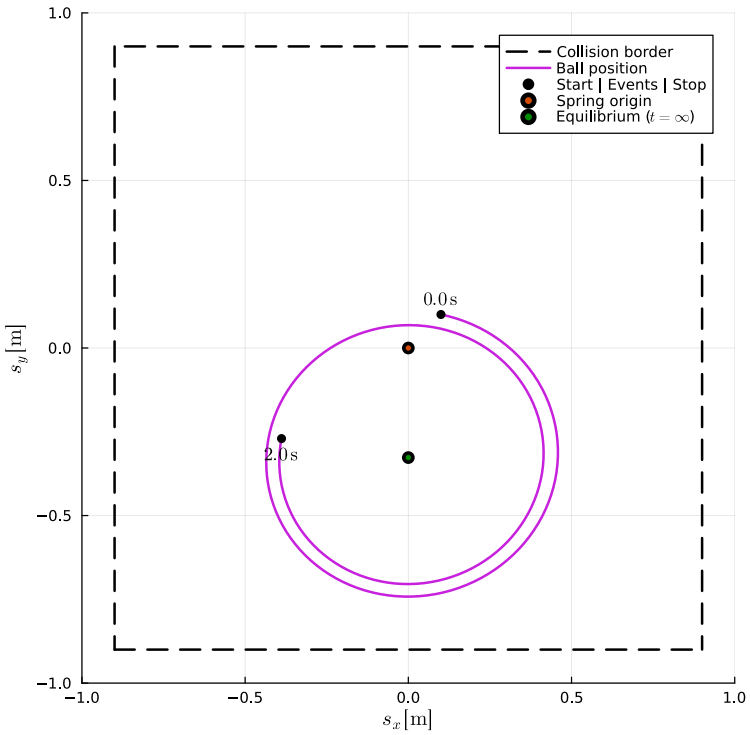


Figure 6.5: Testing trajectory for the bouncing ball with spring. The BB oscillates around its equilibrium position (green) and exhibits only slight damping (radius decreases).

6.7.1.2 Results & discussion

We start by investigation of the baseline model, the hybrid model trained without any regularization applied (SOL). By investigating Fig. 6.6, we find that the pure hybrid (SOL) and the frequency-regularized models (SOL+FRQ+OSC), as well as the damping- (SOL+DMP+OSC) and the stability-regularized models (SOL+STB), converge in similar minima. While all trajectories appear very similar, the regularized trajectories converge a little faster at the very beginning at around step 500. We summarize, that there is no major difference in the median loss trajectories during training.

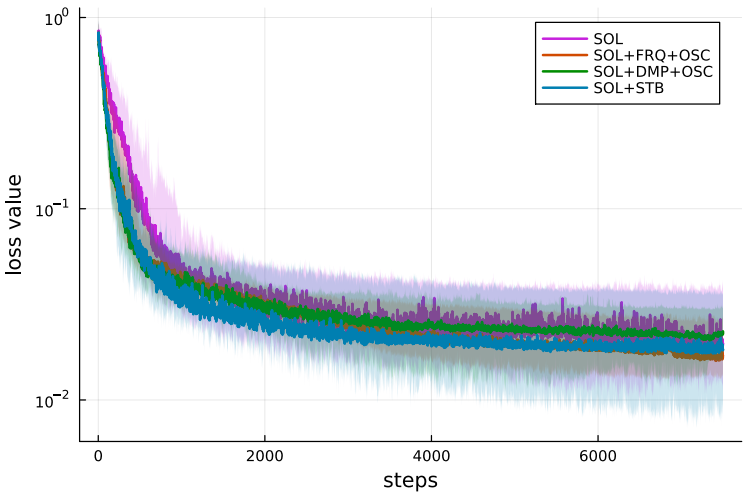


Figure 6.6: Comparison of the convergence of the hybrid model without eigen-informed training (SOL) and multiple kinds of eigen-informed training (SOL+...) by incorporation of different system properties during the training scenario. The semi-transparent ribbons show the 25th and 75th percentiles.

The major goal of eigen-informed regularization in this experiment is to enhance generalization of the model, so the ability to perform on unknown data. Comparing the training loss trajectory to the testing loss (s. Fig. 6.7), we find a different scenario: The pure hybrid model performs the worst on testing data, whereas all experiments incorporating eigenvalue information converge to better minima. This exemplifies, that eigen-informed regularization, based on additional system knowledge besides data, can be used to enhance the generalization capabilities of hybrid models.

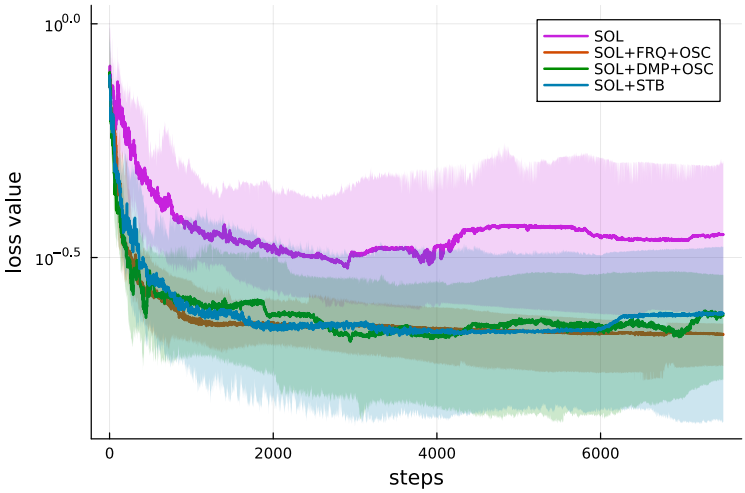


Figure 6.7: Comparison of the hybrid model without eigen-informed training (SOL) and multiple kinds of eigen-informed training (SOL+...) by incorporation of different system properties on test data. The semi-transparent ribbons show the 25th and 75th percentiles.

These observations can further be quantitatively verified by examination of Tab. 6.2.

Table 6.2: Results of the eigen-informed regularization experiment for training and testing data.

Loss	Median (train)	P_{25} | P_{75} (train)	Median (test)	P_{25} | P_{75} (test)
SOL	0.019	[0.011, 0.04]	0.354	[0.241, 0.509]
SOL+FRQ+OSC	0.017	[0.014, 0.027]	0.216	[0.185, 0.229]
SOL+DMP+OSC	0.022	[0.016, 0.03]	0.238	[0.173, 0.29]
SOL+STB	0.018	[0.009, 0.036]	0.24	[0.141, 0.334]

To summarize, we showed that eigen-informed regularization is capable of improving the generalizability of hybrid models and is therefore a valuable tool for challenging hybrid modeling tasks. A more detailed study with focus on NODEs instead of hybrid models, showing that eigen-informed regularization is able to improve convergence speed, the accuracy of the learned solution and is even capable of improving the numerical properties of the learned right-hand side compared to the ground truth model, can be found in the own work Thummerer and Mikelsons [121]. However, the validation of these methods for large-scale industrial applications is future work (see Sec. 9.2), and we limited ourselves to exemplify the methods for the recurring bouncing ball example, here.

Contribution 10 *We introduced* eigen-informed *regularization, which corresponds to an explicit regularization technique for the optimization of (neural) ODEs and HUDA-ODEs. Here, the eigenvalue positions are computed in a differentiable way within the objective function, and different kinds of regularization terms, based on stability, stiffness, oscillation capability, frequency, damping, and more, can be added to the optimization objective. This technique can be used to improve training in various ways, like e.g. incorporation of frequency information or stabilization.*

6.7.2 Events

We can easily exemplify event chattering with the bouncing ball, because it is a common issue especially for this system. Because the system is damped, the ball velocity successively decreases and the frequency of collisions with the ground increases to the point that event handling fails because of numerical issues[7]. Common implementations for the bouncing ball therefore introduce a minimum velocity, that if reached, stops the bouncing ball and statically puts it on the ground. For this experiment, this artificial measure is intentionally not implemented to motivate the benefits of regularizing event indicators, which implicitly corresponds to regularizing the *frequency of events*. For this purpose, a challenging scenario for the bouncing ball is picked in which it is rolling just above the ground, from left to right. This corresponds to *almost* triggering the collision event along the entire simulation trajectory. It is important to note, that such scenarios are common especially in modeling mechanical systems, for example, mechanical stops are often modeled by events, but

[7]For example, the dynamic solver step size reaches the minimum value between events or the event time instant (zero crossing) cannot be located exactly.

are not reached during proper operation of the system (e.g. industrial robot). The simulation trajectory of the rolling bouncing ball can be investigated in Fig. 6.8.

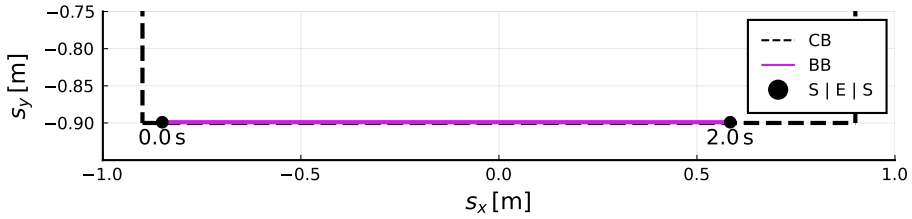


Figure 6.8: The bouncing ball (BB), rolling from left to right, but *not* triggering any events, because it is not (but almost) touching the collision border (CB). Start, events, and stop ("S | E | S") are marked with black dots and time annotations. Even though this may seem like a very simple scenario at first glance, our simulation model reaches its limits here.

The experimental setup is summarized in Tab. 6.3. The solver is parameterized to allow a maximum of $1,000$ events in order to achieve a reasonable training performance.

Table 6.3: Setup for the event regularization experiment.

Attribute	Value
Training strategy	Single Shooting (SB)
Loss function	MAE (s_x, s_y) + $l_{EFR}(\alpha = 10^{-6})$
Loss evaluation points	$\{0.0\,\text{s}, 0.1\,\text{s}, ..., 1.9\,\text{s}, 2.0\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam w. exp. decay
	($\eta_{start} = 0.001$, $\eta_{stop} = 0.0001$, $\lambda = 0.0005$)
Architecture	PSa (SOP)
ANN layout	$6 \rightarrow 8$ (tanh)
	$8 \rightarrow 8$ (tanh)
	$8 \rightarrow 2$ (identity)
Number of training steps	1,000
Number of repetitions	100

6.7.2.1 Hypothesis

In general, the investigated scenario seems challenging. If the damping of the bouncing ball is learned too strong, the bouncing frequency increases very fast, leading to a challenging to simulate model. This also affects the calculation of the gradient. Further we assume, that the loss function value can become very small, even if lots of events are triggered — the BB colliding at high frequencies should be able to match the target trajectory well. Indeed, we can summarize that events in general are not taken into account by the loss function, as long as the target trajectory is met. Incorporation of event information seems promising here, because finding a solution with a reasonable number of triggered events is an implicit requirement

in hybrid modeling. Often, this requirement can already be fulfilled implicitly by specifying a solution trajectory — *implicitly* in terms of the event information is *contained* in this trajectory. If this is not the case — like in our example here — then event information must be *explicitly* included in the cost function.

6.7.2.2 Results & discussion

First, we investigate the loss curves along the training process, see Fig. 6.9.

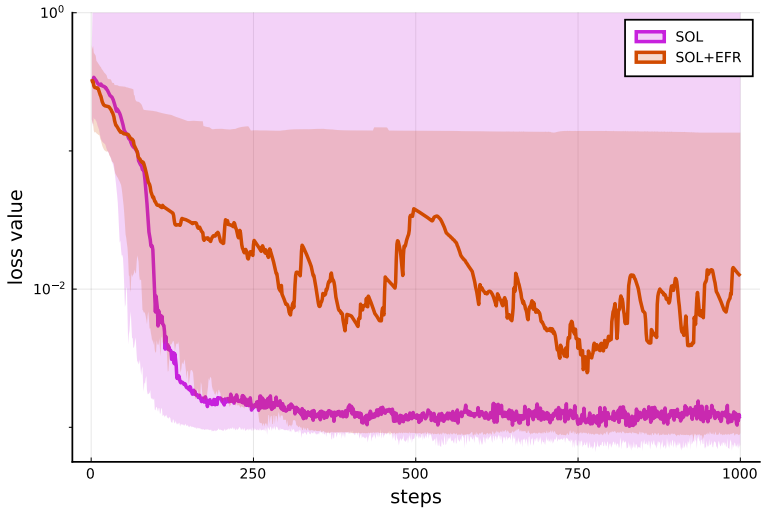
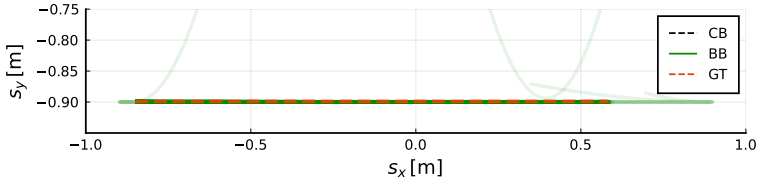
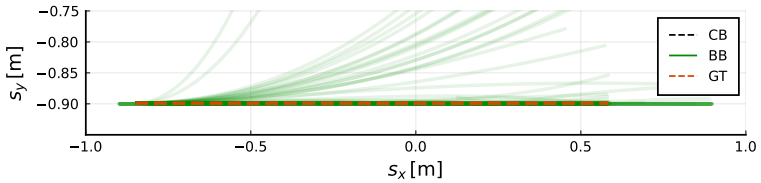


Figure 6.9: The median loss function value for the different regularization strategies compared. The semi-transparent ribbons visualize the 25th and 75th percentiles. The vertical axis is limited, the 75th percentile for SOL reaches infinity.

We find that the optimizer is able to find a better median minimum for the loss function without regularization. Further, the unregularized loss function converges faster. However, investigating the ribbons for the percentiles, we find that the 75th percentile for the loss SOL is infinity (axis are limited), motivating a deeper investigation. It stands out positively, that the 25th percentiles are very similar for both approaches. This means that both approaches can deliver similarly good results when filtered (slightly) for the better results. These observations cover with the solution trajectories, see Fig. 6.10. Here we find, that (almost) all trajectories of SOL are able to provide a good result for the "rolling" ball, whereas we find more outliers for the trajectories corresponding to SOL+EFR.



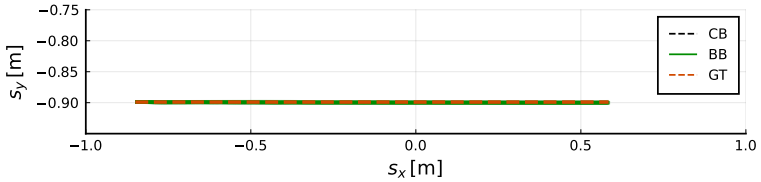
(a) *SOL:* The "bouncing" ball without regularization.



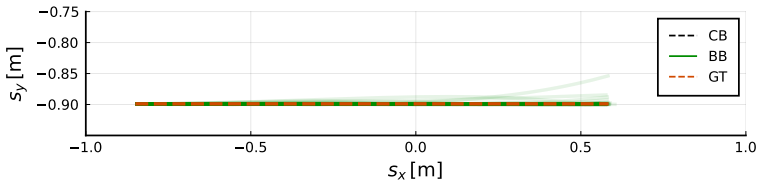
(b) *SOL+EFR:* The "bouncing" ball with event frequency regularization.

Figure 6.10: Comparison of the bouncing ball (BB) with and without event regularization by plotting the simulation trajectory of each repetition (spaghetti plot). Further, we find the collision border (CB) and the ground truth (GT) trajectory.

However, Fig. 6.10 presents an overly positive picture, as unsolvable trajectories cannot be shown. Only plotting the left half of the distribution (0th to 50th percentile) shows, that *good* runs are indeed capable of finding a proper trajectory for both approaches, see Fig. 6.11.



(a) *SOL:* The "bouncing" ball without regularization.



(b) *SOL+EFR:* The "bouncing" ball with event frequency regularization.

Figure 6.11: Comparison of the bouncing ball (BB) with and without event regularization by plotting the simulation trajectory of each repetition (spaghetti plot). Further, we find the collision border (CB) and the ground truth (GT) trajectory. Only the left half of the distribution is plotted (0th to 50th percentile).

The corresponding quantitative improvements are given in Tab. 6.4.

At first glance, the median loss value is smaller for the approach not using regularization. If we investigate the 25th percentile, however, the loss values are almost the same, showing that the better runs lead to equally good results. The 75th percentile is even more interesting, because it is showing "Inf" for the not regularized approach (SOL), indicating that the system cannot be solved. This is because the solver terminates the simulation, caused by event chattering. Interestingly, the very best minimum found is even better for the hybrid model involving event regulariza-

Table 6.4: Loss values for the event chattering experiment, after training steps 500 and 1,000.

Experiment	Step	Median	P_{25} | P_{75}	Min
SOL	500	0.001173	[0.000911, Inf]	0.000359
SOL+EFR	500	0.037437	[0.000904, 0.140089]	0.000373
SOL	1,000	0.001204	[0.000747, Inf]	0.000379
SOL+EFR	1,000	0.012451	[0.000907, 0.135688]	0.000258

tion. In summary, even if the median loss of the not regularized approach is better, the better runs (25th percentile) lead to equally good results for both approaches, while for the worst runs (75th percentile) the system is not even solvable without regularization. This already demonstrates a clear benefit of the regularization method.

However, we recall that the original problem is the difficult solvability caused by event chattering, and we therefore investigate the number of triggered events in Fig. 6.12.

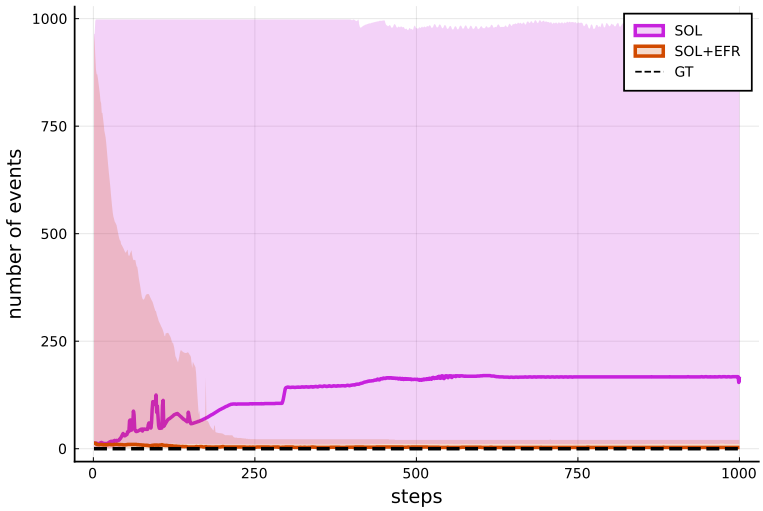


Figure 6.12: The median number of triggered events for the scenario with (SOL+EFR) and without event regularization (SOL) compared. The semi-transparent ribbons show the 25th and 75th percentiles of multiple repetitions. The ground truth (GT) features no events (black-dashed). Note, that the maximum event count is limited to 1,000.

First, we observe that the median event count for the SOL approach is significantly higher, compared to SOL+EFR. Further, the 75th percentile of SOL+EFR event count is rapidly falling to a small value far below the median SOL event count. The 75th percentile of SOL remains on a very high value of $\approx 1,000$, which is the limit for triggered events before termination of the solving attempt. To visualize especially the results for the SOL+EFR event count, we can further zoom in, s. Fig. 6.13.

Here, we find a very distinct difference between the two approaches. While the unregularized approach SOL features a very high median event count of > 160,

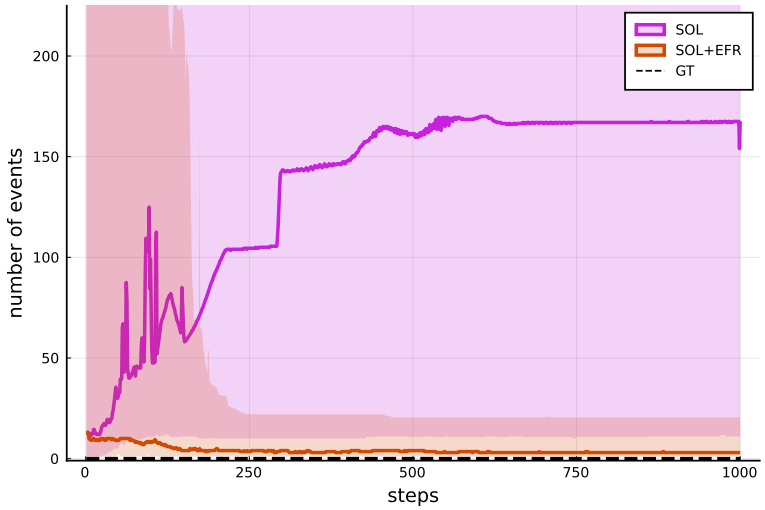


Figure 6.13: The median number of triggered events for the scenario with (SOL+EFR) and without event regularization (SOL) compared (zoomed in). The semi-transparent ribbons show the 25th and 75th percentiles of multiple repetitions. The ground truth (GT) features no events (black-dashed).

the regularized approach SOL+EFR is able to learn for significantly smaller event count of 3 in the median. We further find, that the 25th percentile for the unregularized loss is even worse compared to the median of the regularized approach, which demonstrates the superiority of the regularization. For the bouncing ball, neither the median nor the percentiles of the loss value and the event count decrease during training in case of SOL. This indicates that a badly initialized hybrid model, that is triggering many events, is not able to recover by itself without the appropriate regularization applied. This strongly motivates the application of the proposed regularization strategy for systems suffering from event chattering, not only to *improve* finding a solution, but even to *find* a proper solution in the first place. The corresponding quantitative results are given in Tab. 6.5.

Table 6.5: Event count for the event chattering experiment, after training steps 500 and $1,000$.

Experiment	Step	Median	P_{25} | P_{75}
SOL	500	161.0	[10.75, 977.75]
SOL+EFR	500	4.0	[1.0, 20.5]
SOL	1,000	167.5	[11.0, 979.0]
SOL+EFR	1,000	3.0	[1.0, 20.5]

To summarize, common loss functions do not take the event count into account. While this does not necessarily cause issues, we find that for challenging scenarios — even if they look trivial at first glance — hybrid models exhibit very high event counts, even if they are able to learn for a solution that leads to a small loss value. However, unintentionally high event counts, referred to as event chattering, slow down the simulation performance significantly and impede proper gradient determination. As a result, the original simulation and training process terminates because of numerical issues during handling of events, whereas the event-informed training

keeps the number of events reasonable. For systems that can not be trained because of issues during event handling, the proposed regularization is capable of providing a valuable workaround, but requires determination of new hyperparameters for the gain of the regularization term.

Contribution 11 *By introducing a regularization strategy for event instances based on the metric of the* estimated average event frequency*, we are able to reduce the amount of (unintentionally) triggered events. Doing so, we can heal hybrid models that suffer from event chattering.*

6.7.3 Errors

Before regularizing errors in our recurring hybrid model, the bouncing ball, we need to *define* error indicators first. As for the previous examples, we want to develop a hybrid model that learns for nonlinear air friction. For every friction force, the direction of the force is opposite to the direction of motion. This is what we expect from a well-designed model and therefore incorporate this relation in form of assertions. We observe, that exactly two forces are acting on the bouncing ball: The gravitational force in y-direction, and the friction force by air. To ensure that air friction fulfills this core principle, we can express this relation in two assertions:

$$sgn(v_x) \cdot sgn(a_x) \leq 0 \tag{6.36}$$

and

$$sgn(v_y) \cdot sgn(a_y + g) \leq 0. \tag{6.37}$$

Note, that we add the gravitational acceleration g in y-direction to compensate for the (intended) gravitational acceleration and obtain the fraction of acceleration that is actually caused by air friction. We can confirm, that for $\phi \in \{x, y\}$ and different signs for v_ϕ and a_ϕ these statements are true. Further, the edge case of one of the terms being exactly zero leads to the entire expression being zero, which again mirrors our intension. However, if both v_ϕ and a_ϕ share the same sign, the assertions get violated. We note, that because we are checking for ≤ 0, we directly match the error indicator definition that rates values > 0 as error. This makes the transition from assertion to error indicator straightforward.

However, to obtain an error indicator $\boldsymbol{\delta}$ that further rates the *intensity* of an error, we need to extend the assertions by a *distance measure*, in this case the absolute acceleration by air friction:

$$\boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} = \begin{bmatrix} sgn(v_x) \cdot sgn(a_x) \cdot \|a_x\|) \\ sgn(v_y) \cdot sgn(a_y + g) \cdot \|(a_y + g)\| \end{bmatrix} = \begin{bmatrix} sgn(v_x) \cdot a_x \\ sgn(v_y) \cdot (a_y + g) \end{bmatrix}. \tag{6.38}$$

Equipped with an error indicator $\boldsymbol{\delta}$ we can proceed with the hypothesis. As usual, we summarize the experimental setup in Tab. 6.6.

6.7.3.1 Hypothesis

Because the introduction and regularization of error indicators corresponds to incorporation of additional physical knowledge, we assume that this leads to benefits during training, like faster convergence and/or finding a better local minimum. However, it is important to clarify that this is not the main goal of the error indicator

Table 6.6: Setup for the error regularization experiment.

Attribute	Value
Training strategy	Growing Horizon (SB)
Horizon start	0.1 s
Horizon loss threshold	0.01
Horizon increment	0.1 s
Horizon length	2.0 s
Loss function	MAE (s_x, s_y) + $l_{ERR}(\alpha = 0.01, \beta = 0.0)$
Loss evaluation points	$\{0.0\,\text{s}, 0.1\,\text{s}, ..., 1.9\,\text{s}, 2.0\,\text{s}\}$
Loss value normalization	Min/Max
ODE solver	Tsit5
Optimizer	Adam w. exp. decay
	($\eta_{start} = 0.001$, $\eta_{stop} = 0.0001$, $\lambda = 0.0005$)
Architecture	PSa (SOP)
ANN layout	$6 \rightarrow 8$ (tanh)
	$8 \rightarrow 8$ (tanh)
	$8 \rightarrow 2$ (identity)
Number of training steps	5,000
Number of repetitions	100

experiment. Many simulation models are not solvable at all if assertions are thrown, resulting in the termination of the training process. For such models, this is not only a measure of improvement, but necessary. We therefore want to investigate the average error, and expect that the hybrid model with error indicator regularization features a significantly reduced error count.

6.7.3.2 Results & discussion

We start by investigation of the loss trajectory over time, compare Fig. 6.14.

We find that the median loss curve of the regularized model is significantly improved (starting from around step $1,000$) and converges to a better local minimum. The corresponding quantitative results, which show improvements in every respect, can be investigated in Tab. 6.7.

Table 6.7: Losses for the error regularization experiment, after training steps $2,000$ and $5,000$.

Experiment	Step	Median	P_{25} | P_{75}	Min
SOL	2,000	0.130902	[0.083375, 0.162733]	0.020759
SOL+ERR	2,000	0.00425	[0.003445, 0.005743]	0.002618
SOL	5,000	0.004609	[0.003436, 0.006287]	0.001959
SOL+ERR	5,000	0.003186	[0.002575, 0.003874]	0.001633

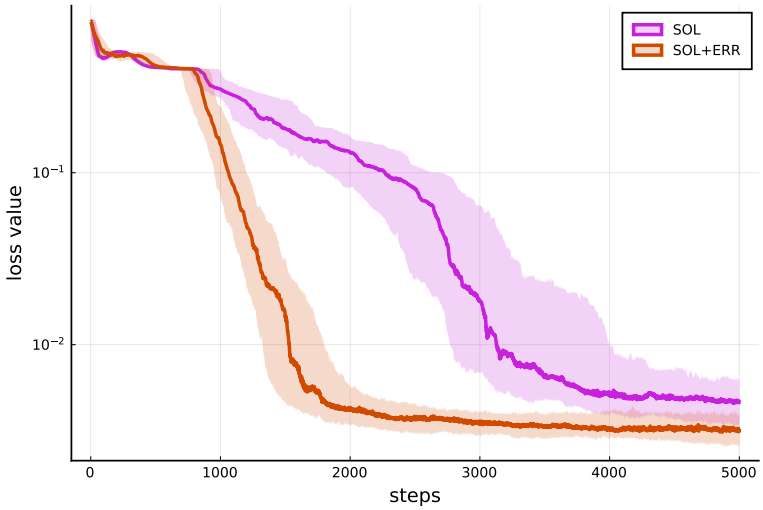


Figure 6.14: The median loss function values for the error regularization experiment, comparing the case with (SOL+ERR) and without regularization (SOL). The semi-transparent ribbons visualize the 25th and 75th percentiles.

The actual benefit of the proposed method — the ability to resolve model assertions that may terminate simulation and training — is shown in Fig. 6.15.

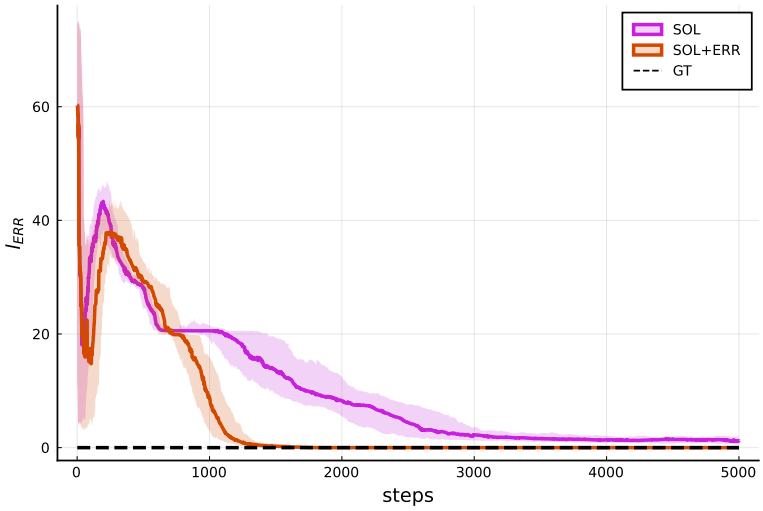


Figure 6.15: The median error loss l_{ERR} for the error regularization experiment, comparing the case with (SOL+ERR) and without regularization (SOL). The semi-transparent ribbons visualize the 25th and 75th percentiles. Further, the ground truth (GT) is given (black-dashed), which exhibits no errors.

Here, we find the median value for l_{ERR}, showing the sum of the positive error indicators, for both scenarios. We see, that the regularized approach is able to cancel out the positive error indicators very fast at the beginning of the training (around step $1,500$), while the unregularized approach improves over the course of training, however, is not able to learn for a model that does not violate the model errors. This impressively shows, that also generalization of the hybrid model can be improved, as this very basic core principle (friction acts in opposite direction to motion) is not satisfied by the learned model without regularization applied. Further, it is important to note, that for more complex models it is no longer necessarily the case

that they remain solvable despite the error violations, making this measure even more relevant.

Contribution 12 *We introduced the concept of error indicators, that not only signal the violation of modeling assumptions, but also* rate *the* intensity *of errors. For hybrid models including model assertions, we showed that regularizing error indicators is not only necessary if the model suffers from unintendedly thrown assertions, but can even improve training convergence for the considered use case. To conclude, we find that also error definitions may contain physical assumptions, and can be interpreted as measure to intuitively incorporate such knowledge into a ML pipeline. This corresponds to a human-driven approach to incorporating prior physical knowledge, because defining assertions feels very intuitive to most model engineers.*

Chapter 7

Real world

Research ends at the latest after the discussion of a few example applications (experiments), which are often kept simple for the sake of easy comprehensibility. Naturally, this is exactly where the work for a development engineer in the field begins. As experience shows, and as anyone who has ever been in this situation can certainly confirm, there are still a few stumbling blocks between research and applying a method in the industrial application. Because this work claims being *industry relevant*, this dedicated chapter is intended to highlight further aspects that are relevant for the practical application of the methods shown.

7.1 Introduction

At the time of writing, the most obvious border between research and application in the field is the interface between the well-established world of (industrial) simulation modeling and the rapidly changing world of ML. It is a matter of fact, that the landscape of available tools either focuses on physics-based simulation models or machine learning. Even if, for example, there are efforts to retrofit ML to long-established simulation tools, tool development is based on requirements from a different era and therefore the new ML requirements form a major obstacle. One exception is *Dyad* — formerly *JuliaSim* — (JuliaHub), which is quite new and features the great advantage of being developed based on requirements from both worlds. It enjoys great popularity in the field of modeling chemical processes, especially in the pharmaceutical industry, but reaches out to new application domains like heating, ventilation, and air conditioning (HVAC) [127] or jet engines [128]. However, there are still open challenges that need to be addressed within the next years to make it as relevant for a broad variety of domains, like e.g. Modelica is today.

The lack of a common language and a common tool for SciML in engineering practice raises the question of the *first step* that makes it possible to exchange simulation models of an industry-relevant size and complexity between the two worlds. Since DEs are considered in this work, the most obvious way would be to exchange the equation system between the modeling tool and the ML environment (e.g. Julia). Unfortunately, there is currently no tool that supports the detailed examination or export of the generated equation system[1]. For example in object-oriented modeling,

[1]Some tools, like for example *Dymola* (Dassault Systèmes), allow inspecting the system via flattened Modelica code (MOF-file export), but this is not required to be complete (e.g. symbolic Jacobians for iterating the algebraic loops).

e.g. with *Dymola* (Dassault Systèmes), *SimulationX* (ESI Group) or *Impact* (Modelon, Inc.), the index of the modeled DAE system is reduced and the resulting ODE system is solved, but the modeler has no access to the final ODE system. At the same time, classical simulation software does not offer access to machine learning features like *backpropagation* (AD) that are crucial for any SciML application. This strongly hinders hybrid modeling and hybrid model training with industrial models within typical development software in the field. Even if systems of equations *could* be exported, further challenges are the efficiently dealing with algebraic loops (and the required Jacobians) and *dynamic state selection* (switching between different sets of equations at runtime to avoid singularities), see Sec. 9.3.

Another approach, and the method used in this work, is the use of FMI as an exchange format between the modeling and ML world. To make this working, an interface between the proposed HUDA-ODE and FMUs must be derived (s. Sec. 7.3). This does especially include an implementation for SA over FMUs. Based on that, hybrid models including FMUs, so called neural/universal FMUs [129], can be derived (s. Sec. 7.5) and special challenges regarding training can be investigated. Further, the big picture of a development cycle for neural FMUs in industry can be drawn in Sec. 7.6 and a corresponding software implementation is briefly highlighted in Sec. 7.7. Finally, the methods proposed in this chapter are not exemplified as in the previous chapters, but are applied within the final validation.

Because for now we only discussed which architectures for HUDA-ODEs are possible (s. Cha. 3), the logical next step is to investigate *meaningful* architectures. In the following, we investigate architectures for HUDA-ODEs that are driven by real-world requirements and are applicable to neural FMUs as well.

7.2 Useful architectures for HUDA-ODEs

Based on the introduced generic architectures in Cha. 3, some specific architectures for typical application scenarios are investigated in the following. Please note, that any of the proposed architectures is fully expressed by a special occupancy pattern of the connection matrix $\boldsymbol{W}$ within the *PSD* architecture.

In many hybrid modeling applications, the starting point is a simulation model that contains the physical system to be modeled or at least a significant part of it. To achieve good coverage, the physical model is assumed to consist of a discontinuous ODE, so a state space, event condition, and event affect function, as well as an additional output function. All architectures have in common, that we only learn for a *continuous* physical effect within the *discontinuous* system model. This means, the ANN is used to learn for the right-hand side or outputs, and not for the event condition or event affect function. Whereas this is also possible from a technical perspective (e.g. see Chen *et al.* [16]), it is unclear if this is meaningful for large scale simulation models and should be considered open research. In summary, the goal is to compensate for an unknown (and not modeled) physical effect by deploying a FFNN modifying states, derivatives, or outputs. Different setups are compared below.

7.2.1 Sequential: Transformation of dynamics

The sequential architecture can be used to correct the system dynamics, compare Fig. 7.1. Here, the state derivatives of the physical system ($\boldsymbol{s}_a$) are corrected by a FFNN ($\boldsymbol{s}_b$). This correction may be performed, depending on the use case, based on the actual state derivatives alone, but it often makes sense to provide additional signals (especially the state of the system $\boldsymbol{x}$ and the system input $\boldsymbol{u}$) as further input to $\boldsymbol{g}_b$.

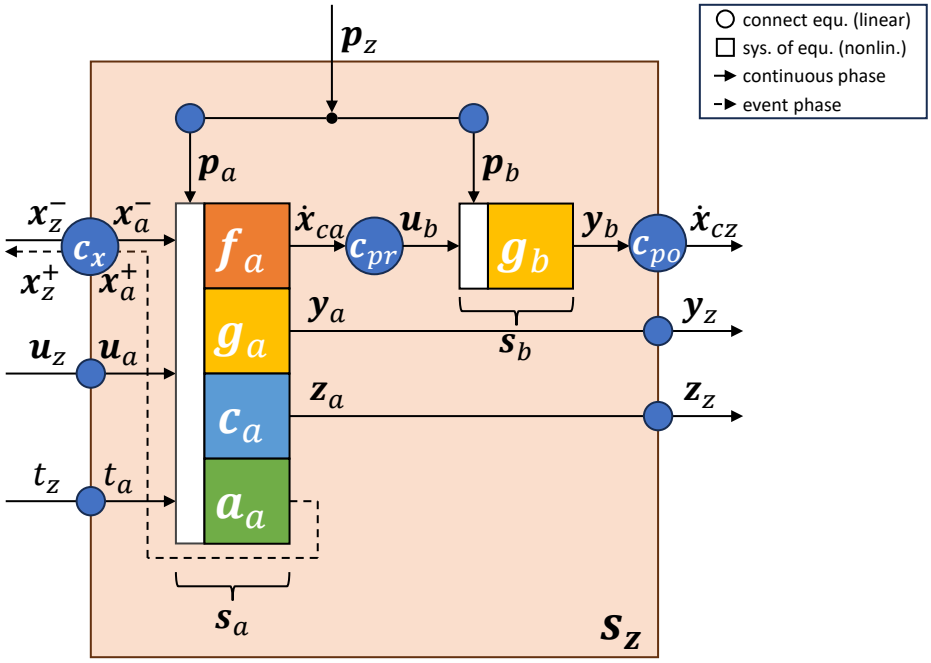


Figure 7.1: Example for a sequential architecture on derivative (*dynamics*) level: The ANN (here a FFNN) $\boldsymbol{g}_b$ is used to *correct* the given dynamics $\dot{\boldsymbol{x}}_{ca}$ from the physical model right-hand side $\boldsymbol{f}_a$ after pre-processing via $\boldsymbol{c}_{pr}$. The corrected dynamics are then post-processed via $\boldsymbol{c}_{po}$ and the hybrid model dynamics $\dot{\boldsymbol{x}}_{cz}$ are obtained. During event handling, a linear system of equations must be solved because of the linear connection $\boldsymbol{c}_x$, to obtain the new hybrid model state $\boldsymbol{x}_z^+$ that is passed on to the solver. However, in practice it is often the case that $\boldsymbol{c}_x(\boldsymbol{x}_z) = \boldsymbol{x}_z$ (so an identity mapping), resulting in $\boldsymbol{x}_z = \boldsymbol{x}_a$, which makes solving the system of equations obsolete.

Mathematically, this procedure corresponds to chaining the functions $\boldsymbol{f}_a$ and $\boldsymbol{g}_b$ for the calculation of the state derivative. From an architectural perspective, the model is constructed by applying the architecture S between the right-hand side of the physical model $\boldsymbol{f}_a$ and the FFNN $\boldsymbol{g}_b$. Further, the functions to define and handle the discontinuity are forwarded, so the hybrid model *inherits* the event definition of the physical model. In summary, this architecture is especially useful if the physical effect can be expressed by *transformation* of the existing dynamics. This is for example the case, if nonlinear physical effects are linearized during modeling and the missing nonlinearity shall be learned by a ANN.

7.2.2 Sequential: State pre-processing

Finally, the sequential architecture can also be used to correct the system state *before* the state derivative is determined, see Fig. 7.2. This can be useful, for example, if there are static (and maybe nonlinear) errors in the state (e.g. due to offsets and/or incorrect coordinate transformation).

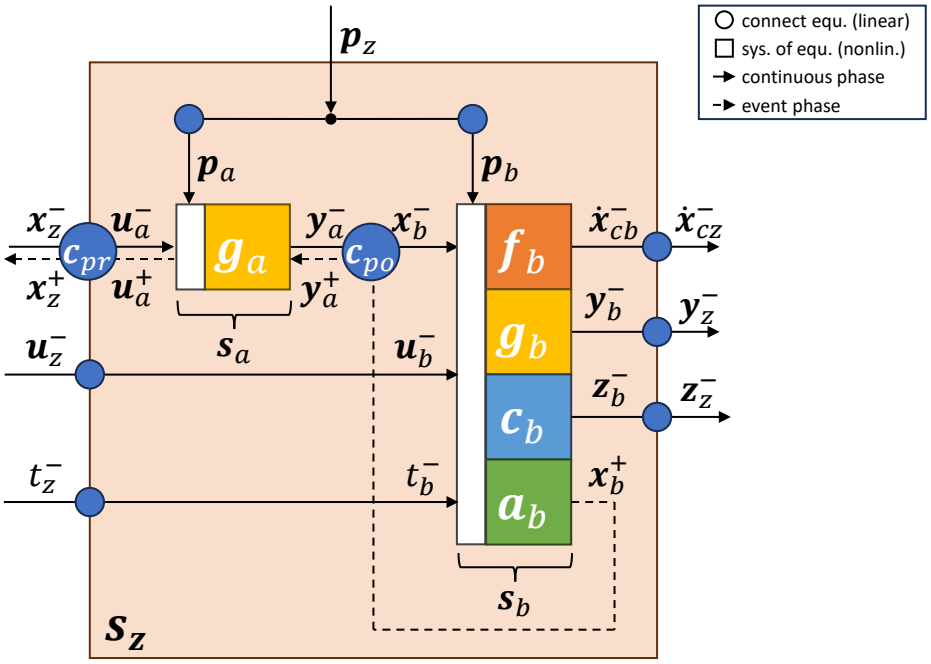


Figure 7.2: Example for a sequential architecture on state level: The FFNN $\boldsymbol{g}_a$ is used to *correct* the given system state $\boldsymbol{x}_z^-$ before passing it on the physical model $\boldsymbol{s}_b$. During event handling, the affect function $\boldsymbol{a}_b$ determines a new state after event $\boldsymbol{x}_b^+$ that needs to be propagated through the function $\boldsymbol{g}_a$ *backwards*. If $\boldsymbol{g}_a$ is invertible, this requires inverting $\boldsymbol{g}_a$, and otherwise, solving of a nonlinear system of equations.

In this case, the FFNN $\boldsymbol{s}_a$ is evaluated before the physical model $\boldsymbol{s}_b$, so the state of the system is transformed. The corrected system state $\boldsymbol{x}_b^-$ is then used in the physical system to calculate the dynamics of the system $\dot{\boldsymbol{x}}_{cb}^-$. A little different from the sequential architectures already introduced, this procedure corresponds to chaining the functions $\boldsymbol{g}_a$ with $\boldsymbol{f}_b$, $\boldsymbol{g}_b$, $\boldsymbol{c}_b$, and $\boldsymbol{a}_b$, because all are affected by the modified system state. This results in not only applying a serial pattern between $\boldsymbol{g}_a$ and $\boldsymbol{f}_b$, but all the functions of $\boldsymbol{s}_b$. For physical systems that contain events, event handling is more complex compared to the serial architecture manipulating the derivative, see also Sec. 3.4.4. This is due to the fact that the new state must be propagated backwards through the subsystem $\boldsymbol{s}_a$ (the function $\boldsymbol{g}_a$, respectively), which generally results in solving a nonlinear system of equations.

As a final note on the sequential architectures, both concepts — transformation of dynamics *and* state pre-processing — can be applied at the same time, which was originally performed in Thummerer *et al.* [92].

7.2.3 Sequential: Transformation of outputs

Whereas the previous approaches directly influence the ODE solution, another option is to only correct the algebraic output equation $\boldsymbol{g}_a$, without actually affecting the ODE right-hand side $\boldsymbol{f}_a$, compare Fig. 7.3.

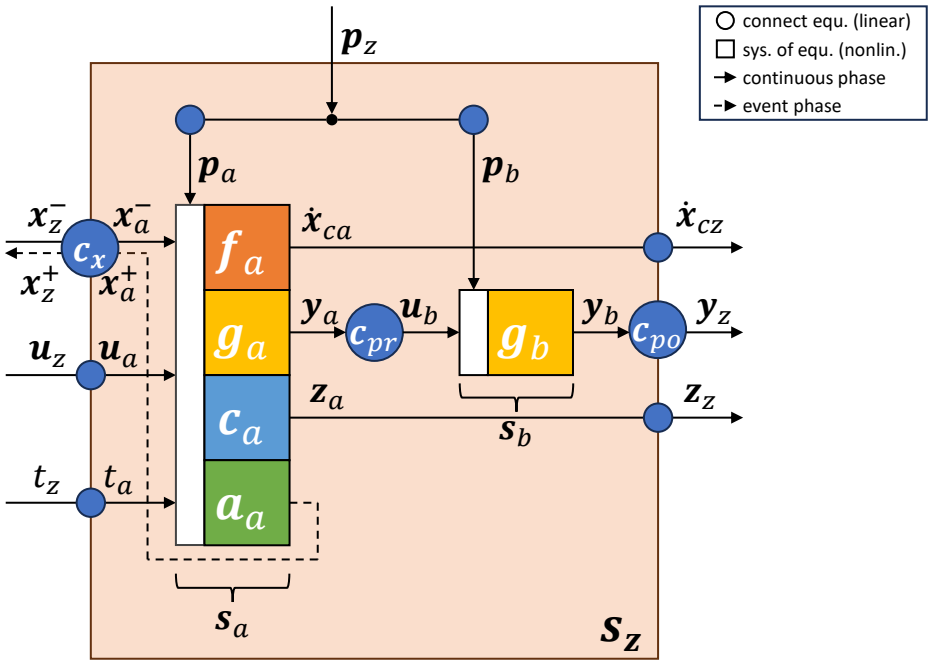


Figure 7.3: Example for a sequential architecture on output level: The ANN (here FFNN) $\boldsymbol{g}_b$ is used to *correct* the output of the physical model $\boldsymbol{y}_a$ after pre-processing via $\boldsymbol{c}_{pr}$. The corrected output is then post-processed via $\boldsymbol{c}_{po}$ and the hybrid model output $\boldsymbol{y}_z$ is obtained.

Similarly to the previous architectures, this procedure corresponds to chaining the functions $\boldsymbol{g}_a$ and $\boldsymbol{g}_b$ for the calculation of the output of the hybrid model. In addition, the hybrid model completely *inherits* the functions that define the right-hand side and the events of the physical model. Beneficially, because the output does not affect the state trajectory, no differentiation is required through the ODE solver. As a consequence, the model can be simulated once to record the outputs, and the ANN can be trained in isolation. This, of course, has a major positive impact on computational performance. During training, no further evaluation of the simulation model is required.

What seems like a major improvement at first glance comes with a significant disadvantage: Although errors can still be compensated for, usually the resulting hybrid models do not generalize well. For example, if the position of a mechanical system should be corrected, but the source of deviation is a force or momentum (in general something that influences the system before *integration* and not algebraically), the function describing the deviation is learned — the position offset — instead of the *source* for the deviation — which is an unmodeled force or momentum. This corresponds to an overfitted model that only performs well in the scenario to which it was fitted. In general, this approach does not yield the intended solution for a hybrid modeling problem, however, in very special applications with sufficient large training datasets or if learning an algebraic relation is actually intended (for

example, the source of error is an algebraic coordinate transform), this architecture could be a reasonable choice.

7.2.4 Parallel: Add dynamics

In the parallel architecture at dynamics level, both submodels share the same state (or parts of the same state), see Fig. 7.4 Furthermore, the input or the time can also be fed into both submodels if the function to be learned depends on the input or the time. Both submodels calculate the system dynamics (or a part of it), both results are then combined to form the overall dynamics (e.g. via addition or concatenation).

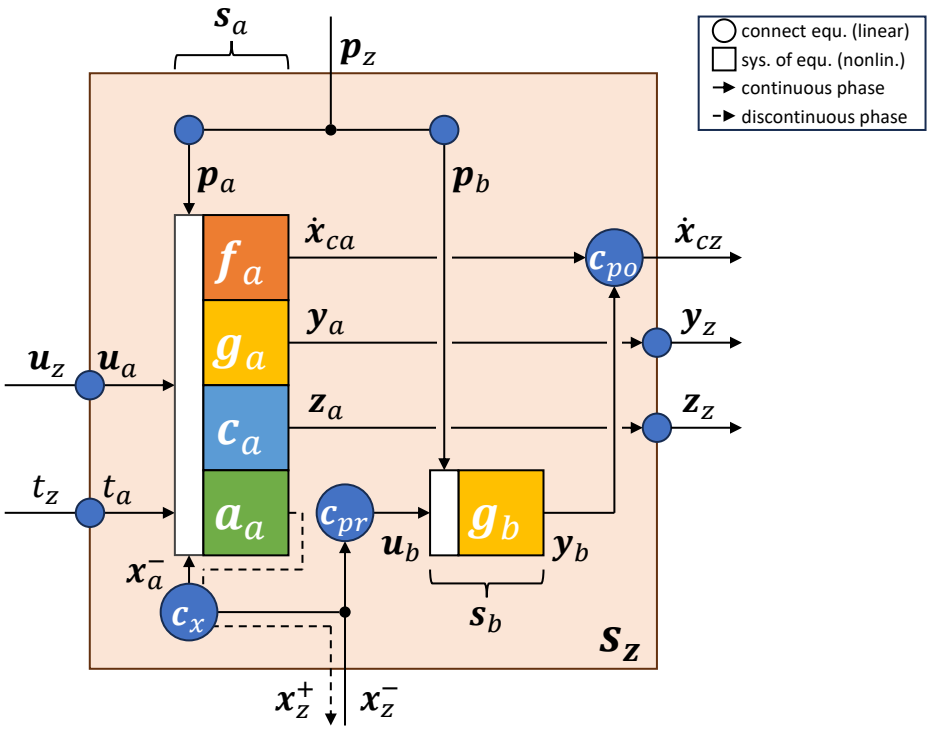


Figure 7.4: For the parallel architecture on dynamics level both submodels, the physical model $\boldsymbol{s}_a$ and the FFNN $\boldsymbol{s}_b$, share the same input state $\boldsymbol{x}_z^-$ (or parts of the same state). Both submodels compute their own state derivative, that is finally merged (e.g. by addition or concatenation) to obtain the hybrid model state derivative $\dot{\boldsymbol{x}}_{cz}$. In case of an event, a linear system of equations must be solved to obtain $\boldsymbol{x}_z^+$.

A parallel setup is especially useful if the dynamics to be learned can be *added* to the existing (physical model) dynamics. This is for example the case if forces or torques are not modeled as part of the corresponding force or torque equilibrium equations. In the example of a mechanical system, the actual force or torque equilibrium does not match the state derivative directly, but the acceleration, which is in general linearly dependent on the force (via the mass) or torque (via the inertia). So adding fractions of acceleration to the state derivative allows for modification of equilibria of forces and torques.

For event systems, the challenges for finding a valid state for after the event $\boldsymbol{x}_z^+$ apply here as well and might require to solve a nonlinear system of equations for every event instance, see Sec. 3.4.4.

7.2.5 Gates: Controlled dynamics

In Thummerer *et al.* [83] the author proposed an architecture based on the introduced *PS* architecture, that allows for training either a serial or a parallel connection of signals (or both) by introducing signal control gates, see Fig. 7.5. Here, even if the FFNN ($\boldsymbol{s}_b$) is fed by the state derivative $\dot{\boldsymbol{x}}_{ca}$ only, in practice it will often be useful to additionally extend the input by the state $\boldsymbol{x}$, input $\boldsymbol{u}$ and/or time t (for explicitly time dependent systems). The obvious advantage of this approach is that it is not required to decide beforehand how the ANN is structurally incorporated.

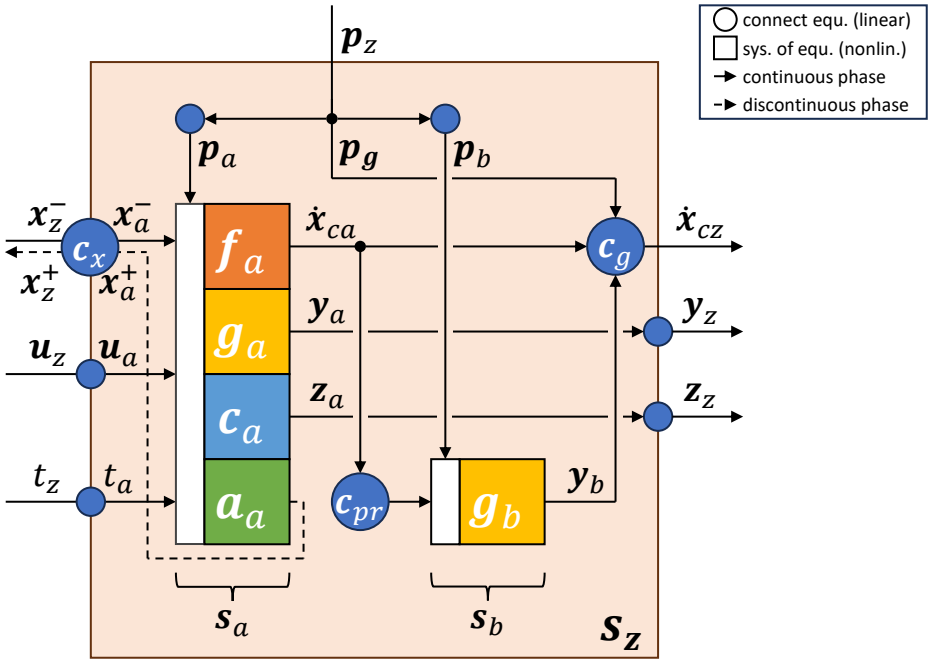


Figure 7.5: In the architecture with gates, the connection equation $\boldsymbol{c}_g$ depends on the parameter vector $\boldsymbol{p}_g \subseteq \boldsymbol{p}$. Accordingly, this connection can be changed during parameter optimization. The connection equation $\boldsymbol{c}_g$ controls the contribution of $\boldsymbol{s}_a$ and $\boldsymbol{s}_b$ to the dynamics of the overall hybrid model.

In contrast to the static *PS* architecture, a subset of entries $\boldsymbol{p}_g$ of the connection matrix $\boldsymbol{W}$ are not set statically, but are only initialized and optimized together with the parameters of the ANN during training. This subset consists of the parameter $\boldsymbol{p}_{ga} \subseteq \boldsymbol{p}_g \subseteq \boldsymbol{p}$, that controls the influence of the physical submodel $\boldsymbol{s}_a$, and $\boldsymbol{p}_{gb} \subseteq \boldsymbol{p}_g \subseteq \boldsymbol{p}$ that controls the influence of the FFNN respectively. Both parameters together determine the impact of the state derivatives by the two subsystems on the overall derivative, as discussed in Sec. 4.5.2.

Multiple interesting border cases can be distinguished for the values of $\boldsymbol{p}_{ga}$ and $\boldsymbol{p}_{gb}$:

Static system For $\boldsymbol{p}_{ga} = \boldsymbol{0} \wedge \boldsymbol{p}_{gb} = \boldsymbol{0}$, the hybrid model state derivative $\dot{\boldsymbol{x}}_{cz}$ is zero and the system is static over time. Whereas this seems only of limited use at first glance, this setup can be used as starting point to obtain a system that can be solved under guarantee by every numerical solver. Successively opening the gates can be a suitable strategy to deal with poor initialization.

Physical model dynamics For $\boldsymbol{p}_{ga} = \mathbf{1} \wedge \boldsymbol{p}_{gb} = \mathbf{0}$, the hybrid model state derivative $\dot{\boldsymbol{x}}_{cz}$ is fully determined by the physical submodel $\boldsymbol{s}_a$. The ANN $\boldsymbol{s}_b$ has no influence on the system dynamics.

Sequential dynamics For $\boldsymbol{p}_{ga} = \mathbf{0} \wedge \boldsymbol{p}_{gb} = \mathbf{1}$, the hybrid model state derivative $\dot{\boldsymbol{x}}_{cz}$ is fully determined by the ANN $\boldsymbol{s}_b$, that is fed by the physical model $\boldsymbol{s}_a$. This equals the sequential architecture on system dynamics level, the ANN is able to *transform* the state derivative of the physical model $\dot{\boldsymbol{x}}_{ca}$.

Parallel dynamics For $\boldsymbol{p}_{ga} = \mathbf{1} \wedge \boldsymbol{p}_{gb} = \mathbf{1}$, the hybrid model state derivative $\dot{\boldsymbol{x}}_{cz}$ is determined by both submodels. In this case, the function learned by the ANN is *added* to the state derivative from the physical model $\dot{\boldsymbol{x}}_{ca}$.

This architecture can be motivated by the case that only little may be known about the effect to be learned and, in particular, how this effect interacts with the existing system.

7.3 Express FMUs as HUDA-ODEs

Although a mathematical class of DEs for hybrid modeling has been investigated so far, FMI describes a software format for the exchange of simulation models. Consequently, a bridge must be established here between the mathematical functions defined for the HUDA-ODE and the standardized software functions in FMI.

In the following, only FMI version 2, the most widely used, and version 3, the latest release and version to be used in the future, are investigated. For function calls that are available in FMI 2 and 3 as well, the placeholder `X` is used and can be replaced by `2` or `3` respectively. The FMI version 1 is not further investigated because it is outdated and has no practical relevance anymore.

7.3.1 ME-FMUs

Because the mathematical pattern of the HUDA-ODE almost matches the one of a ME-FMU, ME-FMUs can be expressed as HUDA-ODEs with only minor adaptions. The necessary measures are introduced below without going deeply into software-related aspects, for example, the function patterns are simplified in order to enhance readability.

Continuous state and derivative
The definition of continuous state $\boldsymbol{x}_c$ and continuous state derivative $\dot{\boldsymbol{x}}_c$ match the ones of FMI and can be directly accessed via the corresponding functions. This way, evaluation of the right-hand side of the ME-FMU is straightforward:

$$\begin{aligned} \boldsymbol{x}_c &\rightarrow \texttt{fmiXSetContinuousState()} \\ t &\rightarrow \texttt{fmiXSetTime()} \\ \dot{\boldsymbol{x}}_c &\leftarrow \texttt{fmiXGetDerivatives()} \end{aligned} \quad . \tag{7.1}$$

If the FMU further provides inputs and outputs, function calls can be complemented as we will see below.

Inputs, outputs and parameters
The inputs $\boldsymbol{u}$, outputs $\boldsymbol{y}$ and parameters $\boldsymbol{p}$ — identified by the model variable tag `causality` — are defined as real, integer, boolean or string variables in FMI. These variables can be accessed via the FMI functions for getting and setting values. However, FMI defines multiple attributes (e.g. `variability`) for variables and might not allow setting and getting values during every phase of the simulation. The corresponding software functions are:

$$\begin{aligned} \boldsymbol{u}, \boldsymbol{p} &\rightarrow \texttt{fmiXSetT()} \\ \boldsymbol{y} &\leftarrow \texttt{fmiXGetT()}. \end{aligned} \tag{7.2}$$

Here, the type `T` in `fmiXSetT` and `fmiXGetT` refers to:

FMI2 `T` $\in$ { `Real`, `Int`, `Boolean`, `Enumeration`, `String` }

FMI3 `T` $\in$ { `Float64`, `Float32`, `Int64`, `Int32`, ... } (see the FMI specification for a complete list [43])

Discrete state
The core problem that hinders a straightforward use of the proposed methods for HUDA-ODEs for *every* (neural) FMU is the unknown discrete state. The discrete state $\boldsymbol{x}_d$ is part of the FMU and its definition further matches the definition within the HUDA-ODE, however in FMI, the discrete state is not represented in the model description nor directly accessible by function — from a user perspective it is unknown. This complicates integration, whereby different cases for the discrete state can be distinguished:

Known discrete state Even if hidden by default, for some special cases the discrete state might be known from expert knowledge of the modeler. In such cases, and if the FMU exposes and allows for manipulation of the variables that correspond to the discrete state, the discrete state can be accessed using the setter and getter methods, resolving the original problem. However, this is highly uncommon and more unlikely, the larger the simulation models become.

Unknown discrete state The default scenario for almost any FMU is that there is no information about the discrete state. A further distinction must be made, depending on whether the optional features for getting and setting the `FMUState` are available[2], namely `fmiXGetFMUState`, `fmiXSetFMUState`, and `fmiXFreeFMUState`:

`FMUState` **available** In case that get/set features are implemented, the discrete state can be captured and set implicitly as part of the so called `FMUState`. The `FMUState` captures the continuous, as well as the discrete state, along further values like e.g. the initial values for the iteration of algebraic loops. However, because the `FMUState` is a non-standardized memory copy, the actual discrete state cannot be extracted or re-injected with a universal approach. Summarized, the discrete state (via `FMUState`) can be captured and applied again, but neither be read nor be set directly.

[2]Not necessary are `fmiXSerializedFMUStateSize`, `fmiXSerializeFMUState`, and `fmiXDeserializeFMUState`.

Setting/getting the `FMUState` is possible at (almost) any time during the simulation.

To allow for proper SA, an artificial discrete state can be introduced. This *dummy discrete state* is based on a simple worst-case assumption: The discrete state changes at every event instant, and after each event, always a new and unique discrete states is entered. Technically, this can be implemented straightforward by introducing a single discrete state, initializing it with zero, and simply incrementing it at each event instance. This results in a discrete state that *counts* the number of events and allows for the unique identification of every continuous section, however only within a single simulation trajectory. For a different simulation, the discrete state must be assumed to be different again. In addition to incrementing the dummy discrete state at events, the `FMUState` is captured[3] that now corresponds uniquely to the dummy discrete state.

This way, we can perform proper SA, because if we need to revisit discrete states in the past, maybe after multiple events happened, we can use the dummy discrete state to identify and apply the corresponding `FMUState`. This `FMUState` implicitly contains the correct unknwon discrete state — even if we do not know its actual dimension or value.

Nevertheless, this measure is only a workaround. For example, hybrid models can not learn for effects depending on the discrete state by applying the dummy discrete state.

`FMUState` **not available** In case that neither the discrete state nor the get/set state features is present, the discrete state needs to be assumed to not exist. Whereas ignoring the discrete state can work in some cases — note, that this does not hinder proper simulation of the model — it might lead to problems, e.g., wrong gradients during local sensitivity analysis.

Because of the impact of this issue, making the discrete state available directly is one topic within the *OpenSCALING* research project, but not finally solved yet. One possible solution will be the definition of a layered standard to access the discrete state in FMI [126].

Event condition and affect

Handling of events in FMI is separated into pure time events and state events that may additionally depend on time. The event condition function for state events can be expressed as a series of FMI calls:

$$\begin{aligned} \boldsymbol{x}_c &\rightarrow \texttt{fmiXSetContinuousState()} \\ t &\rightarrow \texttt{fmiXSetTime()} \\ \boldsymbol{z} &\leftarrow \texttt{fmiXGetEventIndicators()}. \end{aligned} \tag{7.3}$$

Determination of time events is interwoven with the actual event handling procedure, compare to the FMI pseudocode example [41]. First, the event condition for time events is efficiently encapsulated into the event affect function, so the new

[3] Additionally, we need to capture a `FMUState` for the discrete state $x_d = 0$, so within the first continuous segment before the first event happens. For simplicity, we can do this for t_0.

time event is determined after handling the previous state or time event:

$$\begin{aligned} \boldsymbol{x}_c^- &\rightarrow \texttt{fmiXSetContinuousState()} \\ t^- &\rightarrow \texttt{fmiXSetTime()} \\ t^*, ... &\leftarrow \texttt{fmi2NewDiscreteStates()} \quad \text{or} \\ &\leftarrow \texttt{fmi3UpdateDiscreteStates()} \\ \boldsymbol{x}_c^+ &\leftarrow \texttt{fmiXGetContinuousState()}, \end{aligned} \tag{7.4}$$

where t^* is the time of the *next* time event. The initial time for the first time event is communicated after initialization. The call sequence above is simplified, the detailed software implementation can be found as part of the *FMI.jl* software repository and is neglected here for the sake of understandability.

Another difference is that the information of which event indicator features a change in sign, which we store in $\boldsymbol{q}$ (compare Sec. 3.3.1), is not available in FMI and is stored hiddenly inside the FMU. Within the FMU, the internal representation of the vector $\boldsymbol{q}$ is used to determine the events to be handled. However, for the HUDA-ODE, the vector $\boldsymbol{q}$ can easily be reconstructed by monitoring the event indicators during and around event handling.

7.3.2 CS-FMUs

A CS-FMU can be interpreted as ME-FMU including a simulation environment, meaning a simulation loop including event-handling. However, in general the system state is not known for CS-FMUs. Even if the CS-FMU is internally solved like any other ODE, the FMI hides this detail and provides only a discretized view of the ODE inside. This results in the following:

- The continuous as well as the discrete state of the CS-FMU is unknown. The same applies for the continuous state derivative.
- As for the ME-FMU, the inputs $\boldsymbol{u}$, outputs $\boldsymbol{y}$ and parameters $\boldsymbol{p}$ directly match the corresponding values of the HUDA-ODE.
- The event condition and affect functions, as well as the handling procedure for events, are completely hidden inside the FMU.

However, also CS-FMUs can be expressed as HUDA-ODEs, but for the general case only with inputs and outputs. In theory, even CS-FMUs can be designed to provide interfaces for states and state derivatives, by adding them as system inputs and outputs, but this requires additional effort during modeling and expert knowledge during operation and should not be considered an approach relevant to practice. Besides the mentioned limitations, *FMIFlux.jl* (s. Sec. 7.7) supports CS-FMUs as well.

7.4 SA for FMUs

Equipped with an implementation that allows for inference of FMUs in the framework of HUDA-ODEs, we further want to allow for *hybrid modeling* with FMUs.

For this, we require — besides handling the calls for the inference of the HUDA-ODE — various sensitivities to allow for proper training of FMUs or hybrid models containing FMUs. Whereas these sensitivities can be determined using AD if the HUDA-ODE is available as differentiable code, FMI does not provide all required sensitivities and a workaround for the missing sensitivities needs to be derived.

Within this section, we investigate two cases for SA:

Internal parameter SA determines the sensitivities for parameters, that are *part* of the FMU. This allows, for example, optimization of physical parameters of the FMU by gradient based optimization.

External parameter SA allows for determination of parameter sensitivities, that are *not part* of the FMU. This is required, for example, if the FMU is paired with an ANN as part of hybrid modeling, and parameter sensitivities of the ANN need to be propagated through the FMU.

Depending on the case, internal or external parameter SA, different sensitivities are required. This topic is also investigated in the own work Thummerer *et al.* [126], in which the subsequent investigation is conducted in greater depth. Here, we derive a solution for this in two steps: First, we examine the required sensitivities and second, we present strategies on how these sensitivities can be determined.

7.4.1 Required sensitivities

Determination of the sensitivities necessary is straightforward: For internal parameter sensitivities, we can investigate the methods for SA introduced in Sec. 2.1.3, because this exactly matches the problem definition of determining parameter sensitivities for a parameter-dependent (event) ODE. For external parameter sensitivities, the interfaces of the FMU need to be investigated, and it must be assumed that sensitivities could be propagated through any combination of FMU known variable (state, input, time, ...) and unknown variable (state derivative, output, event indicator, ...) — if there is a meaningful use case for it.

Internal parameter sensitivities
Independent of the actually applied method for SA, the interface of the FMU is statically given, so the same sensitivities are required. In the following, we briefly summarize the steps necessary for simulating a single solver step involving an event, and proceed by investigating these steps on derivative level.

The event condition $\boldsymbol{c}(\tilde{\boldsymbol{x}}, \boldsymbol{p}, \tilde{t})$ is evaluated iteratively, until the corresponding event indicator (that was identified previously) shows a zero crossing. Here, the iterative procedure operates on $\tilde{t}$, however in order to evaluate $\boldsymbol{c}$, $\tilde{\boldsymbol{x}}$ needs to be determined by numerical integration or evaluation of the preliminary solver polynomial. After a termination criterion is met, the event time t^* is found. Based on the corresponding event state $\boldsymbol{x}^-$, the event affect $\boldsymbol{a}(\boldsymbol{x}^-, \boldsymbol{p})$ can be evaluated and the state after event $\boldsymbol{x}^+$ is obtained. As a result, the following derivatives are required:

- The derivatives over the right-hand side $\boldsymbol{f}$, which is used within $ODESolve$, so $\partial\dot{\boldsymbol{x}}/\partial\boldsymbol{x}$ and $\partial\dot{\boldsymbol{x}}/\partial\boldsymbol{p}$.

- The derivatives over the event condition $\boldsymbol{c}$, so $\partial \boldsymbol{z}/\partial \boldsymbol{x}$, $\partial \boldsymbol{z}/\partial \boldsymbol{p}$, and $\partial \boldsymbol{z}/\partial t$.
- The derivatives over the event affect $\boldsymbol{a}$, so $\partial \boldsymbol{x}^+/\partial \boldsymbol{x}^-$ and $\partial \boldsymbol{x}^+/\partial \boldsymbol{p}$.

For the sake of completeness, if inputs $\boldsymbol{u}$ are used, the corresponding partial derivatives w.r.t. $\boldsymbol{u}$ are required, too. The same applies for the time t, if the corresponding functions — $\boldsymbol{f}$, $\boldsymbol{c}$, and $\boldsymbol{a}$ — depend on time directly.

External parameter sensitivities
Because it is unclear how the FMU — or the HUDA-ODE respectively — is integrated in the later hybrid modeling application, various sensitivities need to be considered. For example, ANNs could be incorporated before or after (in terms of a sequential architecture) any of the functions $\boldsymbol{f}$, $\boldsymbol{c}$, $\boldsymbol{a}$ or $\boldsymbol{g}$. To summarize, this captures all Jacobians $\partial \boldsymbol{\gamma}/\partial \boldsymbol{\upsilon}$ between any pair of unknowns $\boldsymbol{\gamma} = \{\dot{\boldsymbol{x}}_c, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{x}_c^+, \boldsymbol{x}_d^+\}$ and knowns $\boldsymbol{\upsilon} = \{\boldsymbol{x}_c^-, \boldsymbol{x}_d^-, \boldsymbol{u}, \boldsymbol{p}, t\}$. This observation makes the requirements for *internal* parameter sensitivities obsolete, because the corresponding sensitivities are fully contained here.

7.4.2 Providing sensitivities

Within this section, we investigate how the required sensitivities are calculated.

Get/Set directional/adjoint derivatives
The FMI standard does not provide functions to access Jacobians directly, however offers an interface for evaluation of JVPs and VJPs. The corresponding function for JVPs is `fmiXGetDirectionalDerivative` (FMI2 and 3) and for VJPs `fmi3GetAdjointDerivative` (FMI3 only). This function interface allows for defining efficient rules for AD. However, not all required sensitivities are accessible by this interface, only the Jacobians $\{\dot{\boldsymbol{x}}_c, \boldsymbol{y}\}$ w.r.t. $\{\boldsymbol{x}_c, \boldsymbol{u}\}$ are mentioned in the standard definition [43]. Even if not explicitly allowed by the standard, some tools like e.g. *Dymola* allow for obtaining further sensitivities, for example w.r.t. parameters.

Finite differences
For the remaining knowns and unknowns — event indicators, parameters, state before/after event, and time — the partial derivatives must be approximated by finite differences (e.g. central differences). Whereas the implementation of finite differences is in general trivial, here, boundaries for the known variables must be respected. FMI provides functionality for specifying minimum and maximum values for model variables, that can be parsed from the model description and be implemented during sampling. However, boundaries can also by dynamic (e.g. depending on the state), which exceeds the possibilities of FMI and again would profit from the proposed error indicators (s. Sec. 6.5). Whereas FMI allows to sample sensitivities within the same time step, a new challenge pops up as soon as sensitivities are required while proceeding in super-dense time, which is investigated in the next paragraph.

Finite differences with `FMUState`
For the special case of SA over an event instance, further measures are necessary in

order to achieve acceptable computational performance. While almost all investigated sensitivities are determined within the same super-dense time step, meaning the known and unknown are investigated for the very same time step without events happening in between, this is not the case for the sensitivity over the discontinuous state change. Note, that $\partial \boldsymbol{x}^+ / \partial \boldsymbol{x}^-$ quantifies how the state *before* event influences the state *after* event. While this results — from a mathematical perspective — in straightforward differentiation of the event affect function, in FMI this results in a new challenge:

- First, there is no built-in function to differentiate the new state w.r.t. the previous state, so sampling (finite differences) needs to be applied. More precisely, one entry of the continuous state before event is perturbed, event handling is performed and the state after event is examined. This needs to be repeated for every state individually.
- Second, FMI does not support setting/getting the discrete state, which directly results in the fact that the discrete state *before* the event is not known and can not be recovered after event handling is performed. Because the state before the event can not be recovered, it is required to start a new simulation from the only time instant where the entire state is assumed known, which is the initial time. However, starting a new simulation for every state and every event instant with state change is computationally infeasible, of course. A workaround for this is to indirectly capture the discrete state by using `fmiXGetFMUState` *before* the event is handled for the first time, and `fmiXSetFMUState` for revisiting the state before the event for the consecutive sampling runs. However, this optional feature must be supported by the FMU.

The presented state is implemented in the Julia library *FMISensitivity.jl*, that is part of *FMIFlux.jl*. Ideas on how the current state of the implementation can be enhanced based on the concept of a *FMI layered standard* [130] — a backward-compatible, standardized extension to FMI — is part of own current research [126].

Contribution 13 *By providing a differentiable interface between HUDA-ODEs and FMI, almost all methods for HUDA-ODEs can be applied to ME-FMUs, if optional FMI features are implemented. The only remaining restriction are hidden discrete states, for which we can apply the workaround of* dummy discrete states*. This directly opens up to pairing ME-FMUs with ML models for hybrid modeling, as investigated in Sec. 7.5.*

7.5 Neural & universal FMUs

As briefly introduced in the introduction, the development of neural FMUs [129] is motivated by the fact that there is a practical component to working with industrial simulation models in addition to the pure mathematical consideration (HUDA-ODEs). While many tools do not support the export of the readable ODE system, many tools still implement FMI. This allows, for example, the export of ME-FMUs which contain the right-hand side of the ODE system in compiled (i.e. not human-readable) form, but allow for almost any evaluation of the system of equations. Although this no longer enables free manipulation of the equation system, it does

allow for extending it, e.g. through state augmentation, addition of further terms or transformation of existing ones. Paired with the previous derivation that ME-FMUs can be expressed as HUDA-ODEs, this immediately enables UAs like ANNs to be efficiently attached to ME-FMUs.

The right-hand side of the ODE can only be modified for ME-FMUs, as no interface for a numerical solver is provided in the CS and SE simulation modes. Anyway, neural FMUs can also be implemented for FMUs in CS (as shown in Thummerer *et al.* [129]) and SE simulation modes, but offer significantly fewer intervention options (s. Sec. 7.3.2) compared to ME and are therefore not further highlighted in the following.

Following the terminology of UDEs, a FMU that is paired with an UA is called a *universal FMU*, or *neural FMU*, if the UA has the form of an ANN. Neural FMUs where first introduced in Thummerer *et al.* [129] by the author. Finally, note that any of the architectures proposed in this work, especially interesting are the ones in Sec. 7.2, are applicable to neural FMUs as well, just by replacing the physical model by a ME-FMU.

Contribution 14 *We proposed the concept of the neural (or universal) FMU, which features the combination of ANNs (or UAs), FMUs, and a numerical solver (if not already part of the FMU). Neural FMUs meet the requirements of real industry demands by allowing for hybrid modeling (and SciML in general) with large scale simulation models including discontinuities and algebraic loops.*

7.6 Development cycle for neural FMUs

As already motivated, it is necessary (for now) to switch tools to being able to do hybrid modeling of industrial scale. Multiple development pipelines are thinkable, as summarized in Fig. 7.6.

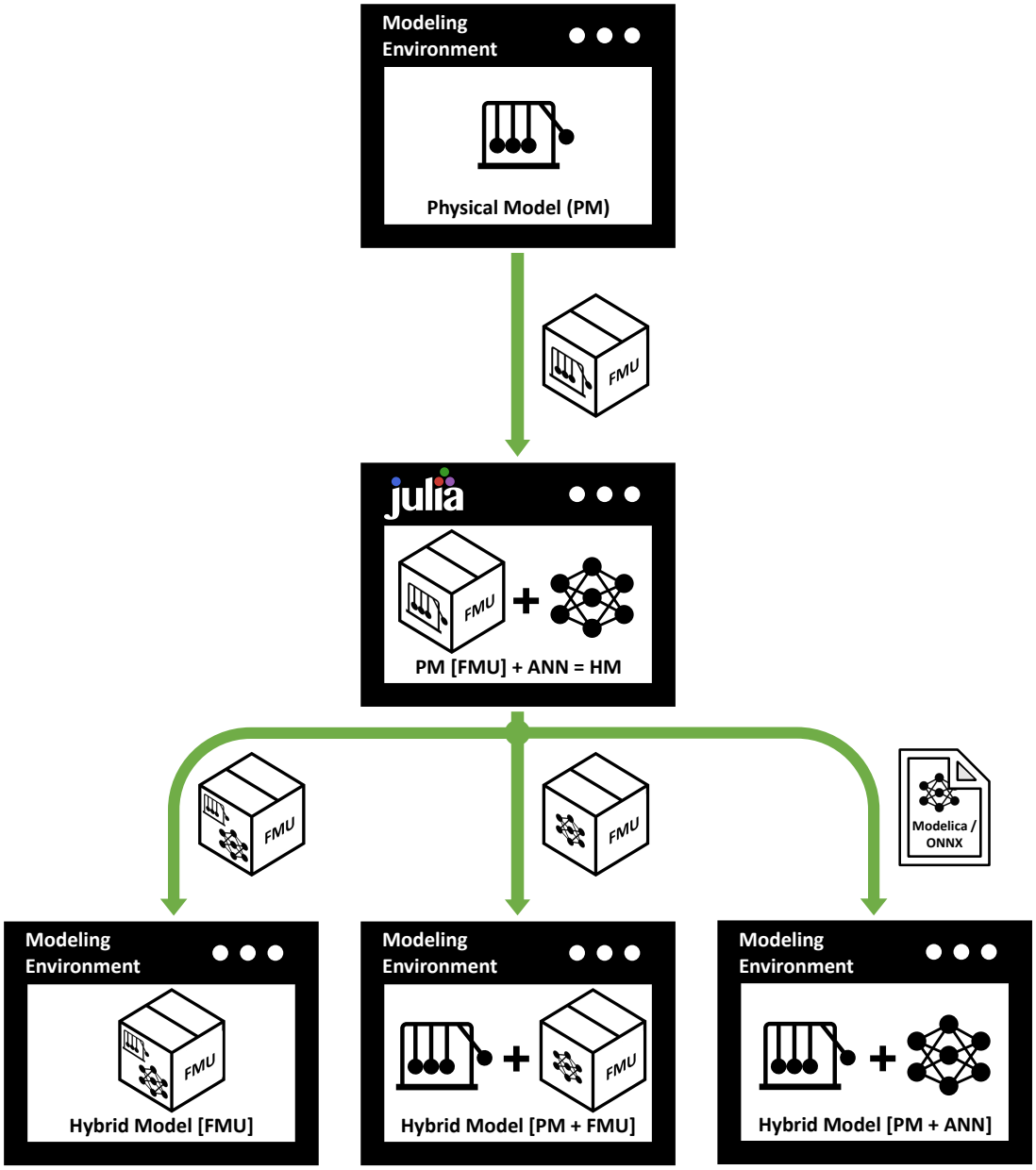


Figure 7.6: Development cycle(s) for neural FMU. This development pipeline was originally proposed in Thummerer *et al.* [83], later updated in Thummerer *et al.* [93] and presented as poster at the *Workshop on Advancing Scientific Machine Learning in Industry* (15th October 2024) at the Technical University Munich.

All the proposed cycles start and end within a conventional modeling tool. First, we start by modeling a simulation model in a modeling tool of our choice. There is no general statement on *how much* of the physical system should be modeled, modeling can end *naturally*, as soon as no further understanding of the physical system is present, or *economically*, if further improvement of the model would be only possible with considerable (and unjustifiable) effort. At this point, the model is exported as FMU. Within Julia, *FMI.jl* can be used to import the FMU. Using *FMIFlux.jl*, the FMU can be extended to a hybrid model by embedding it in a ML model. The hybrid model can be initialized, trained (and regularized) and tested within Julia. After hybrid modeling succeeded, multiple options can be considered:

Hybrid model FMU The hybrid model can be exported as FMU, for this purpose the Julia library *FMIExport.jl*, which is part of *FMI.jl*, was developed. Most modeling tools with FMI export also support the import of FMUs, so the hybrid model can be imported back into the original (or another) modeling tool and integrated into an application. Disadvantageous, the structure of the hybrid model is fixed and statically compiled into the FMU. However, parameters of the model (and therefore the ML model within) can still be changed, because FMI explicitly provides an interface for this.

ML model as FMU Instead of compiling the entire hybrid model into an FMU, only the ML model can be. This way, the original simulation model can be kept in its original form (for example *Modelica* code), while only the ML model has a fixed and compiled structure. This of course requires additional implementation effort for the model interfaces to reconstruct the hybrid model architecture within the original modeling tool. However, this comes with the advantage that modifications to the physical model are still possible, while parameters of the ML model are also changeable within FMI.

Dedicated formats for ML models Instead of compiling the ML model to a FMU, dedicated formats for export and import of ML models can be used. For example, the ML model can be translated to Modelica and reused within Modelica-based tools as in Thummerer *et al.* [93]. Another option is the open neural network exchange (ONNX) format.

Contribution 15 *Neural (and universal) FMUs allow for a workflow to start hybrid modeling within common modeling tools. After extending the model in Julia, featuring the possibilities for high-performance ML, multiple ways to reintegrate the hybrid model into the original modeling environment are available, that provide different benefits. This way, hybrid modeling can be integrated intuitively into existing engineering workflows with only low entry barriers.*

7.7 Custom software libraries

To apply the proposed methods in practice, multiple software libraries where developed, that are all open-source. The three most important are introduced in this section. However, software-related details are deliberately not discussed in this dissertation.

FMI.jl

Figure 7.7: The *FMI.jl* logo [131].

With the goal to bring FMUs to a machine learning environment, the development of *FMI.jl* started. Before being able to combine FMUs and ANNs, the first step is to import FMUs in Julia. Even if at the beginning of writing there was a Julia package for simulation of FMUs available, called *JuliaFMI.jl*, the implementation status was prototypically (the package was archived in the meantime). Therefore, a new project was started in 2021, the development of *FMI.jl*, with the explicit goal to implement the entire FMI standard, starting with FMI version 2.0. Over the past years, the library was extended successively and now supports, besides the FMI standard 3.0, many additional helpful features for inspection, linearization, and plotting of FMUs.

FMIFlux.jl

Figure 7.8: The *FMIFlux.jl* logo [132].

After importing FMUs into Julia is enabled by *FMI.jl*, the overarching goal can be pursued, which is to provide a framework for hybrid modeling with FMUs. Besides the technical challenges — like efficiently combing FMUs and the existing frameworks for SA, including proper handling of state- and time-dependent discontinuities — one major design question is how FMUs could be combined intuitively with ML models from a user perspective. Here, it was decided to allow FMUs being interpreted as layer of ANNs, which allows for very easily combining FMUs and ANNs. In this way, also incorporation of multiple FMUs in parallel or sequence is possible. Based on that, the development of *FMIFlux.jl* started and has been able to serve various challenging industrial applications over the years.

HUDADE.jl

Over time, a wide variety of models were implemented in the course of this dissertation, starting with many physical simulation models, neural FMUs (*FMIFlux.jl*), and further common ML models. In this context, the question arose on how far methodological considerations could be generalized to capture the actual possible range of engineering applications. The result of this reflection is the concept of HUDA-ODEs. The corresponding package *HUDADE.jl* supports definition, as well as combination of HUDA-ODEs and will be the future foundation of the packages *FMI.jl* and *FMIFlux.jl*.

Chapter 8

Validation

The important final step in this work is *validation* — using real, challenging applications from industry. However, validation involving black-box models is not trivial.

8.1 Introduction

For validation in an academic environment, it makes sense to use a hybrid model consisting of a black-box model that can be "opened" — so the equations can be investigated — to check the validity of results. In this case, however, the black-box is actually not a black-box anymore, and there is a risk of using knowledge about the system of equations more or less deliberately. As a result, this would diminish the significance of the validation w.r.t. black-box models in the (engineering) field. On the other hand, if real industrial models are applied, there is a decisive new dilemma: We lose the possibility of investigating the mathematical details of the physical model within the hybrid model, because these are in general not accessible, e.g. because of compilation or reasons of intellectual property (IP). In general, it is not possible for an academic institution to gain full insight into a real industrial model in the field. This presents us with a tricky challenge since neither the approach of validation on an industrial black-box nor on an academic white-box alone is sufficient.

8.1.1 Use cases

To tackle this, we are validating on three different use cases, which have different levels of obscuration with regard to data and model, see Tab. 8.1. First, the electronically commutated motor (ECM), which can be seen as closed-source industrial simulation model, including closed-source data. The pig cardiovascular system (PCS), that is an open-source model trained on closed-source data. And finally the fully open-source example of the selective compliance assembly robot arm (SCARA). The fourth case, a closed-source model trained on open-source data, is implicitly contained within the existing three use cases, does not lead to a stronger statement, and is therefore omitted.

These three use cases allow us to critically validate the various conditions regarding the availability of models and data, and make a good general statement about the applicability and relevance of the methods presented.

Table 8.1: Source availability of models and data used for the validation.

Experiment	Model open-source	Data open-source
SCARA	✓	✓
ECM	✗	✗
PCS	✓	✗

As introduced, we investigate three use cases in the following that share the major restriction that was motivated in the introduction: a very limited amount of real data. The use cases and corresponding models are briefly introduced in the following:

Selective compliance assembly robot arm (SCARA) A robot arm that can draw on a sheet of paper is investigated in Sec. 8.2. The "real" system suffers from stick-slip-friction, that is not part of the simulation model, leading to a significant deviation between model and data of the real system. This use case corresponds to hybrid modeling for accuracy improvement. Data generation requires operation of the real system, which requires a prototype. Measuring the exact tool center point (TCP) position is challenging and requires additional effort. In SciML, we can use other quantities that are easily accessible, like the motor currents, to improve the model prediction.

Electronically commutated motor (ECM) In Sec. 8.3, we investigate a high-resolution simulation model of an ECM driven by pulse-width modulation (PWM) signals, featuring state- and time-events at very high frequencies. The model is part of system development in the industrial field, but it does not meet the requirements for software in the loop (SiL) testing w.r.t. accuracy in its current state. The goal, increasing accuracy, is a typical use case for a hybrid model. Data is very limited here, because this requires operation of an expensive real prototype.

Pig cardiovascular system (PCS) The PCS (s. Sec. 8.4) is a stiff simulation model from the arterial tree of the cardiovascular system of a pig, featuring a spatially distributed artery model and vascular beds modeled by three-element Windkessels. While the simulation accuracy of the model is sufficient, the simulation performance needs to be improved. This is a typical use case for a fast hybrid model surrogate (computational performance). Data is very limited here, because this requires measurements of the real system (patient). Making additional measurements comes with a long list of complications for the patient.

When selecting these use cases, care was taken to ensure that the corresponding models are highly distinguishable with respect to their attributes, as summarized in Tab. 8.2, to validate the broad applicability and robustness of the methods.

8.1.2 Benchmarking

In pure ML, validation is typically performed on established benchmark problems. For image classification, reinforcement learning and other regions of ML there are

Table 8.2: The almost contrary properties of the three investigated use cases for validation.

Property	ECM	SCARA	PCS
Origin	Industry	Academia	Industry
Goal	The prediction **accuracy** shall be increased, because the simulation model is not accurate enough.	The prediction **accuracy** shall be increased, because the simulation model is not accurate enough.	A surrogate of a high-fidelity simulation is needed that features better simulation **performance**.
Type of DEs	Designed as DAE, which is further transformed into an ODE by index reduction.	Designed as DAE, which is further transformed into an ODE by index reduction.	Modeled as PDE, that is transformed into an ODE by applying spatial discretization via finite differences.
States	Very few: 4 continuous time states and an unknown number of discrete states.	Very few: 6 continuous time states and 2 discrete states.	Many: 2,538 continuous states (fine-grid model), but no discrete states.
Inputs	Multiple inputs, e.g. PWM signals by the motor controller, load on the motor, etc.	No inputs, the trajectory to be drawn (text file) can be interpreted as input.	The arterial blood mass flow in the Aorta.
Periodicity	No periodic behavior.	No periodic behavior.	Periodic behavior after a short phase of transient initialization.
Solver	Features a zero eigenvalue for the entire operation space. However, it is solved with a fixed step explicit method (`Heun`).	Stable system, which is solved with the explicit Runge-Kutta method `Tsit5` [133] (error control via step size).	A stiff simulation that requires an implicit solution method (`FBDF`).
Algebraic loops	Unknown, not exposed.	One linear (dimension 2).	Two linear (dimensions 4 and 2) and 10 non-linear (dimension 2) algebraic loops, one for every arterial branch.
Events	State and time, triggered at high frequency, resulting in an event frequency of $\approx 10^4$ 1/s.	State-events, that are not triggered during regular simulation. Time events to sample target positions for the trajectory to be drawn.	State-events, that are not triggered during regular simulation.
Controller	Controlled in the later application (advanced controller), but hybrid model trained open-loop (without controller).	Controlled system model (proportional control), hybrid model trained with active controller.	No controller.
Data	Real data, including noise and external influences.	Synthetic data from a high-fidelity simulation model.	Mix of real and synthetic data. In-vivo measurements for the input, simulation data for non-observable quantities.

benchmarking problems available that allow for a meaningful comparison of different methods. Common datasets are, e.g. *Imagenet* [134], *CIFAR-10* and *CIFAR-100* [135] for computer vision applications or *Gymnasium* [136] for reinforcement learning. Unfortunately, no established benchmarks for hybrid modeling in SciML exist at the time of writing. However, in the following, some candidates — may be for the near future — are listed, together with the reason they were not used today.

SciMLBench Suite The *SciMLBench Suite* [137] seems an interesting option for the near future, but currently only three datasets are available: Two for image classification, one for image noise reduction, but unfortunately none for time series prediction. Therefore, this benchmark was not considered further.

PDEBENCH The benchmark suite *PDEBENCH* [138] provides 11 examples for SciML applications based on PDEs, therefore targeting e.g. PINNs and Fourier neural operators (FNOs). It allows for comparison of SciML methods with classical PDE simulations and pure ML models. The models cover, among others, the Navier-Stokes equations in different variations (e.g. compressible and incompressible), as well as with different dimensionality (1D, 2D and 3D). However, as the name suggests, the suite is aiming at PDEs and is therefore not suitable for the validation of HUDA-ODEs.

SuperBench The *SuperBench* collection aims at validation of SciML models in the context of super-resolution applications, which covers the retrieval of finer details in the domain of computer vision [139]. Unfortunately, this is not the scope required for validation of the methodology introduced within this thesis.

MLPerf™ HPC With a focus on large-scale SciML training applications, *MLPerf™ HPC* [140] allows for benchmarking on high performance computings (HPCs) systems. However, the type and size of the proposed problems (and the corresponding models) is far beyond what is investigated in this thesis, e.g. involving up to 2048 GPUs.

PeN-ODE Benchmark Suite The *PeN-ODE Benchmark Suite* will be a collection of challenging problems and state-of-the-art solution approaches for physics-enhanced neural ordinary differential equations (PeN-ODEs), that are a subclass of HUDA-ODEs (s. Sec. 9.3). Unfortunately, at the time of writing, this is still a part of active development in the *OpenSCALING* research project and will not be publicly available before 2026.

Even if there is no suitable benchmark suite available, the results must be critically validated. For this purpose, we will compare the hybrid models with the baseline of the pure physical model and a pure ML model. We note that a pure ML model will not perform well suffering from a lack of data and short training times, and it would require to change the boundary conditions w.r.t. the amount of training data, training duration, etc. in order to generate any usable results. Changing the amount of data, however, is not possible for the use cases investigated, and we conclude, that the considered applications are not satisfactorily solvable with pure ML. Anyway, we provide the best possible — but still not good — results of the pure ML approach side-by-side to give an assessment of the situation.

8.1.3 Hyperparameter optimization

To reduce the influence of poor initialization and suboptimal hyperparameters, all experiments are subject to hyperparameter optimization, and only the best runs are presented. This corresponds to an industry-oriented way of working: Even if the robustness of approaches plays an important role scientifically, only a single (the best) result is usually relevant for the actual subsequent application in the field. However, we include the examined hyperparameter space and the found hyperparameter set in the appendix. The following hyperparameters are investigated for all experiments:

Optimizer The Adam [98] optimizer is applied with varying parameterization, with step sizes $\eta \in [10^{-6}, 10^{-2}]$ and exponential decay rates for the first and second momentum estimates $\beta_1 \in \{0.99, 0.9\}$ and $\beta_2 \in \{0.9999, 0.999, 0.99\}$. Whereas the default values of Adam perform well across various applications [98], the training of HUDA-ODEs, and neural ODEs in general, which feature special challenges like expensive gradients and small ANN layouts, may require another parameterization.

Loss function As loss function, the MAE or MSE is applied, because the distribution of the measurement data is unknown. Also, because of the relative small amount of training steps, generally known rules of thumbs, such as using MSE for normal distributed data, may not apply here.

Batch duration As introduced in Sec. 5.2, mini-batching requires the separation of the trajectory in small chunks with fixed temporal length, referred to as *batch duration*. We investigate different durations depending on the use case.

FFNN layout Different layouts for the FFNN are compared, featuring widths between 8 and 128 neurons and depths between 2 and 4 layers. All layers use the same activation, possible options are *tanh*, *sigmoid*, *relu* or *leakyrelu*. Note, that because of the linear connections within the *PSDa* architecture, allowing for learning the post-processing parameters after the FFNN, also the last layer can be equipped with an activation featuring saturation.

Gate opening All use cases apply gates, and different initial openings for the gates for the FFNN are applied, reaching from 0 % to 50 %. The gates are optimized together with the ANN parameters during training.

However, some experiments extend this set by additional hyperparameters, that are introduced separately. For hyperparameter optimization, the optimizer *Hyperband* is applied [141]. The optimizer features multiple *brackets*, each with a different weight between exploration (check many different hyperparameter samples, corresponding to the number of samples) and exploitation (check convergence of a specific hyperparameter sample, corresponding to the allocated resources). The brackets feature equal resources, often the execution time like here, and can be executed in parallel. As resource value, we apply $R = 81$, which can be interpreted as the amount of processed training data in seconds. For drawing samples within the brackets, an ordinary random sampler is used. Furthermore, for the reduction factor $\eta = 3$, the default value is applied, which means that the best $1/\eta = 1/3$ of the samples is retained for further optimization within each round and $\eta = 3$ times more resources are allocated for further investigation of the promising samples.

8.1.4 Training

The training process is technically the same as for any other optimization problem. An optimization function is defined, the gradient with respect to parameters is calculated, and the optimizer performs an optimization step based on the gradient information. The loss function, however, contains solving the HUDA-ODE for the hybrid model, and the solution is compared to GT data. The horizon of training data (*resource*) and batch element duration is determined by the outer hyperparameter optimization loop, and the number of training steps results directly from the chosen resource and batch element duration.

All use cases presented have one thing in common: We only use a *single* trajectory of training data to optimize the hybrid model. This features two interesting benefits. First, we drastically highlight that data is very limited — too limited to do proper ML. Second, every *other* trajectory for testing and validation is by design labeled *out-of-distribution* data, because a single trajectory does only result in an infinitesimal narrow distribution. Only points *exactly* on that single trajectory[1] are *in-distribution* data, but are not met in practice — except for the initial state $\boldsymbol{x}_0$. This circumstance allows for critically examining the extrapolation and generalization capabilities of the hybrid model.

[1]To put it very precisely: This is only true for systems with explicit time dependency, because for such systems two different measurements will never share the same trajectory points (state, input, time) in practice. Note, that the ECM, PCS, and SCARA all feature time-dependent inputs. For implicitly time-dependent systems, only the state-input-space (without time) must be investigated to rate in- and out-of-distribution.

We further apply the *LimIter* regularization for all use cases (compare Sec. 6.6), allowing for a maximum of approximately ten times (for the ECM two times) the number of steps it requires to solve the physical simulation model. For the batch elements, we scale the number of allowed solver steps by $1/b$, where b is the number of batch elements, in order to adapt to the shorter time span. The pure physical simulation models vary in stiffness, however they never exceed this individual threshold for any batch element.

8.2 Robotics engineering: Selective compliance assembly robot arm (SCARA)

The example of the SCARA is deliberately chosen as an introduction to validation because, even though it is of academic origin, it is very striking and an ideal platform for presenting the advantages of the introduced approaches. The SCARA example stems from own workshops given at *JuliaCon 2024* [142], as well as similarly at *MODPROD 2024* [143].

8.2.1 Introduction

Robots in general have fascinated people for a long time, and probably these machines are most interesting when they replicate (complex) human tasks. The task of *writing* is fundamental and challenging, and has occupied inventors already a quarter of a millennium ago. *The writer* by Pierre and Henri-Louis Jaquet-Droz and Jean-Frédéric Lescho is probably the first machine (1768-1774) that was capable of writing custom texts [144]. Since then, mechanics have been expanded by electrical engineering and later by electronics and software, and the construction of such writing robots has become a popular student exercise.

Use case
Contrary to the other two validation examples, the SCARA is an artificial use case, meaning the simulation model and data is not based on an existing demonstrator, but is created for the purpose of this example. This offers the advantage that both the model and the results can be examined in greater detail[2], as there are no restrictions of IP.

The SCARA in this application is a robotic arm that writes with a pen on a piece of paper, see Fig. 8.1. The robot consists of two rotary axes for positioning the pen over the paper, and one translational axis for lifting and placing the pen on the paper. By adding this third axis, the robot does not have to draw continuously and is able to interrupt drawing during movement.

The robot arm receives a list of coordinates for the TCP — the position of the pen tip — that are driven to sequentially, one after another. However, the electric motors were chosen very economically in terms of maximum torque, so that although the motor torque is *sufficient* for writing in most areas of the paper properly, a new effect appears at the outer edge of the working space, with the arm almost completely extended: stick-slip friction. Stick-slip friction can be observed from various mechanical systems and is characterized as follows:

[2]We do not need to normalize values, can use axis ticks, and display physical units.

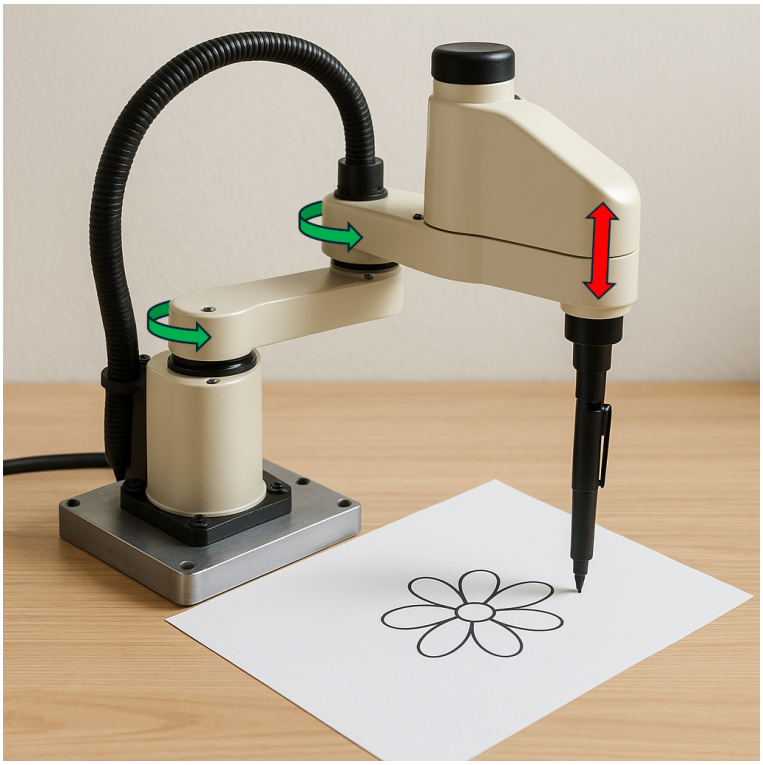


Figure 8.1: The SCARA, drawing with a pen on a sheet of paper. The two rotary axes (green) position the pen on the paper, and the third translational axis (red) is able to lift the pen to stop drawing. This image was generated with the assistance of AI.

Sticking If only *small* forces are applied in a tangential direction (here: a force is applied to the pen in any direction within the paper plane), nothing happens — the body *sticks* to the ground (here: the paper). For small forces, there is an (infinitely) high coefficient of (static) friction.

Slipping If a certain force threshold is exceeded, the coefficient of friction drastically decreases (dynamic friction) and the body suddenly starts to move. In a controlled system, this sudden change often leads to a rapid counter-reaction by the controller, so that the controlled system is quickly slowed down again. In the worst case, the system slows down to such an extent that it reenters the sticking state.

The change between these two states leads to a characteristic oscillation. This phenomenon is not trivial in modeling. Even though simple models for this effect exist — we will investigate one in the upcoming paragraph *Data* — they are not trivial to parameterize.

Due to these difficulties, within the considered use case, this effect is not taken into account during designing the physical model, resulting in a deviation between the data (which includes stick-slip) and simulation model (that does not include stick-slip). The goal of this use case is to build a hybrid model which is able to learn for the not modeled — and challenging to model — stick-slip friction effect based on data.

Physical model
The physics-based simulation model of the SCARA is built in Modelica, using the *Servomechanism* Modelica library [145]. We omit a detailed presentation of the model at the equation level, as the focus of this work is not on physical modeling. Nevertheless, we are introducing the model to the extent that the experiment can be followed. Interested readers can find the complete Modelica model as part of the *FMIZoo.jl* repository. The physical model of the SCARA consists of five subsystems, compare Fig. 8.2:

Trajectory The input trajectory to be drawn by the robot consists of time points and 2D-coordinates. These coordinates are the target positions for the TCP (the tip of the pen) over time. The trajectory is stored in a text file and can be changed between simulations.

Robot mechanics The mechanical subsystem consists of two rotational joints, as well as mechanical links in between. This part of the mechanical subsystem lays entirely in the x-y-plane. A translational axis is attached to the last link, which allows the pen to be moved in the z-direction and thus place it on or lift it from the paper.

Inverse kinematics A robot kinematics model calculates the TCP position and orientation based on given joint angles and the extrusion for translational joints. The inverse kinematics of the robot does the opposite, it calculates the required joint angles and extrusions for a given TCP position and orientation. For the SCARA, joint angles are determined for a given pen tip coordinate. Note, that the pen orientation around the x- and y-axes is constant, and the orientation around the z-axis is obsolete[3] in this application.

Motors The electric motors generate torque depending on the motor current, and are used to operate the SCARA mechanics. Further, the motors are extended by gearboxes to apply higher torques to the robot arms.

Motor controllers The motors are controlled by a simple proportional controller, determining the motor voltage based on the deviation of the motor angle.

[3]The pen writes the same independent of its rotation around the z-axis.

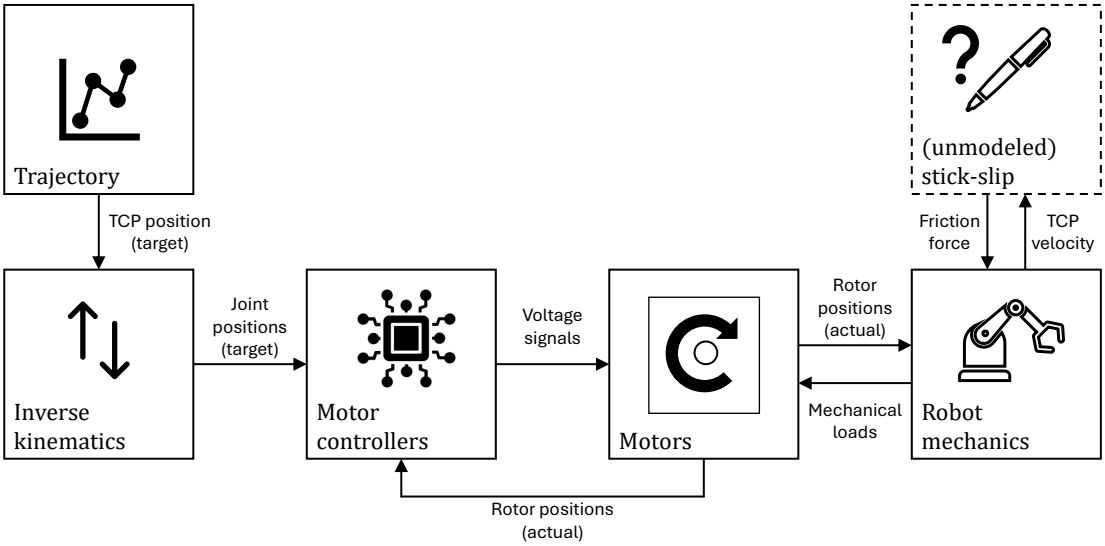


Figure 8.2: The five subsystems of the SCARA, as well as the unmodeled sixth subsystem: the stick-slip. The arrows in between the subsystems are labeled with the corresponding signals. Image is adapted from the *MODPROD 2024* slides [143].

Data

For the SCARA, data is generated artificially by simulation of a more detailed model of the system, including the stick-slip friction model. A common approach is to approximate stick-slip friction with a continuous curve as given in Fig. 8.3. This stick-slip-model is from the *Modelica PlanarMechanics* library [146].

The considered curve is parameterized by the following four parameters:

v_a The maximum velocity of the sticking body, so the maximum velocity for which the adhesion friction applies. For larger velocities, the body enters a meta-state between pure sticking and slipping.

v_s The minimum velocity of the sliding body, so the minimum velocity for which the sliding friction applies. For smaller velocities, the body enters a meta-state between pure slipping and sticking.

μ_a The adhesion friction coefficient, giving the maximum ratio between adhesion friction and normal force for the sticking phase.

μ_s The sliding friction coefficient, giving the ratio between sliding friction and normal force for the sliding phase.

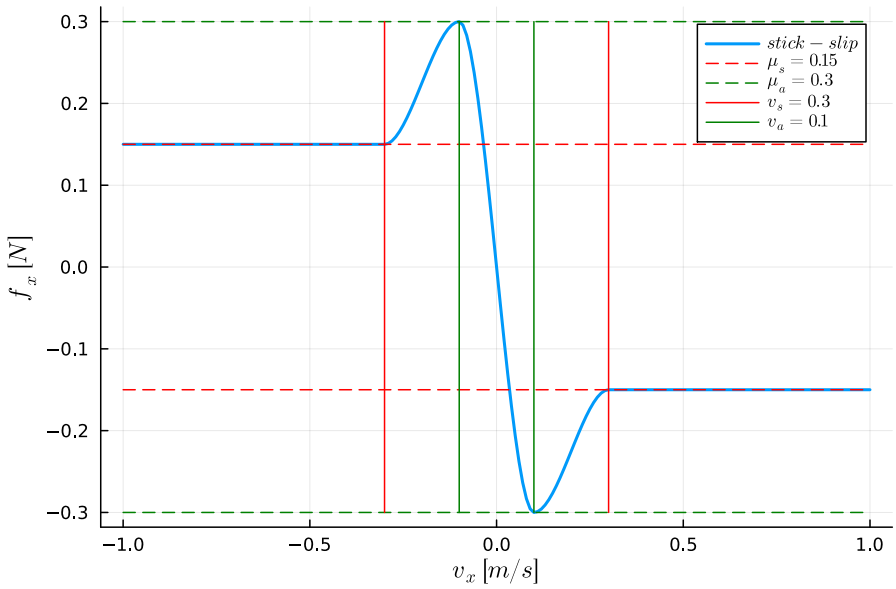


Figure 8.3: The continuous stick-slip-curve in direction x for a normal force of $1\,N$. There are three phases: First, for $v_x \in [-v_a, v_a]$ (in between the solid, green lines) the body *sticks* to the ground. Second, for $(v_x > v_a \wedge v_x < v_s) \vee (v_x < -v_a \wedge v_x > -v_s)$ the body *switches* between sticking and slipping. Third, for $v_x \geq v_s \vee v_x \leq -v_s$ the body *slides* on the ground.

For data generation, this stick-slip curve is integrated into the simulation model and data is generated by simulation of three different scenarios: Writing the words "train" (for training data, see Fig. 8.4), "validate" (for validation data, see Fig. 8.5) and "test" (for test data, see Fig. 8.6). We scale all three words so they have an approximate length of 20 cm, this results in different heights for the letters within the three words, reaching from ≈ 4 cm (validate) to ≈ 6 cm (train) to ≈ 10 cm (test). This provides an excellent basis for testing the extrapolation capability of the hybrid model.

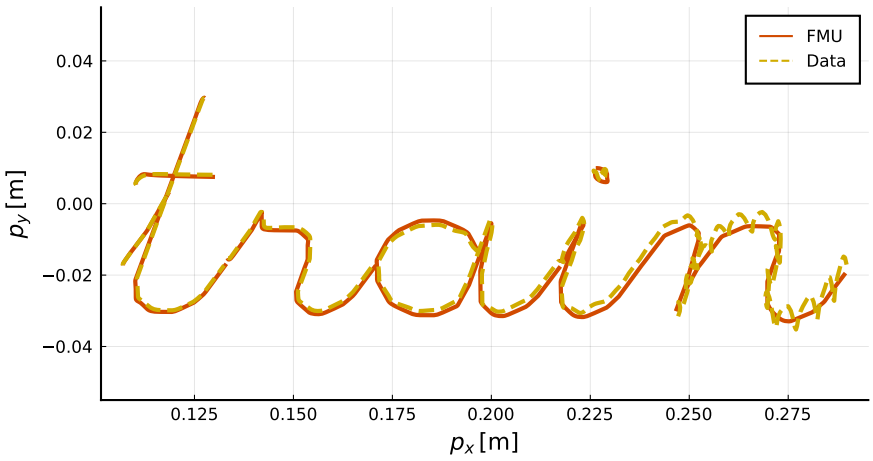


Figure 8.4: Training data (written trajectory) for the SCARA.

Note, that the robot base (where the robot is connected to the ground) is located at the center of the coordinate frame. This means that the robot arm becomes

longer the further to the right we write, and in analogy greater torques are required to overcome friction. We find that the word "train" is written nicely at the beginning, but a slight deviation becomes noticeable at the letter "i", which then clearly transitions into the expected stick-slip effect at the letter "n". As a rule of thumb, we note that stick-slip becomes noticeable at around $p_x \approx 0.2\,\mathrm{m}$, and very clear starting from $p_x \approx 0.25\,\mathrm{m}$.

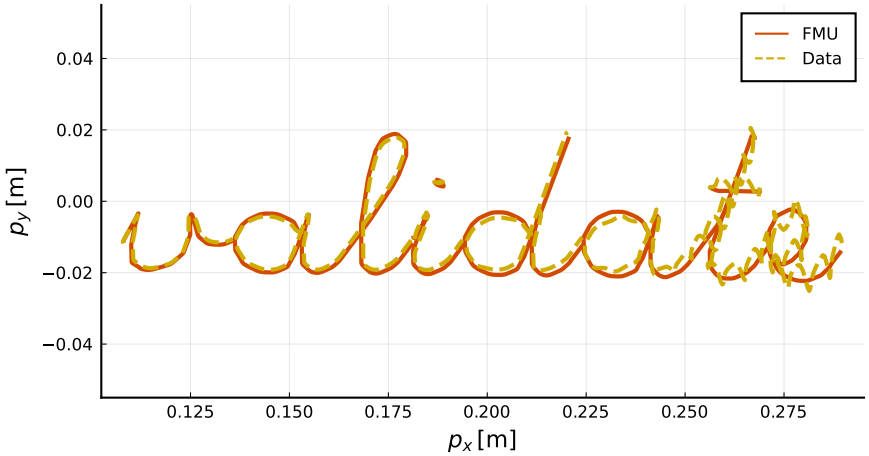


Figure 8.5: Validation data (written trajectory) for the SCARA.

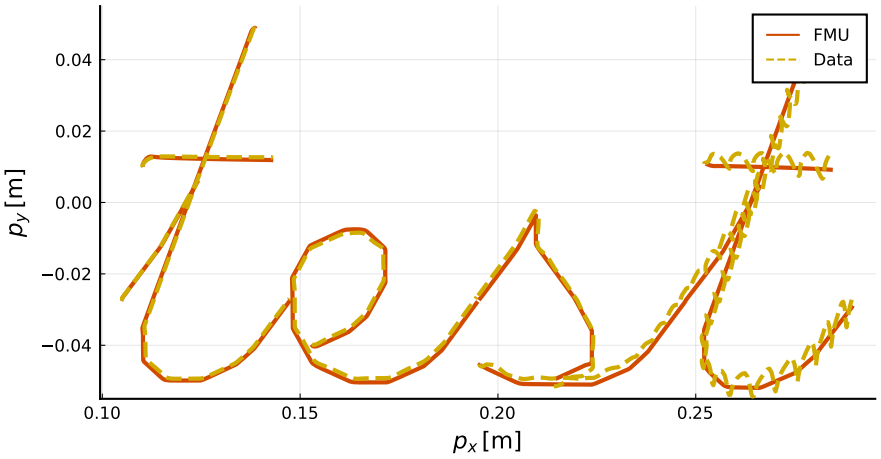


Figure 8.6: Test data (written trajectory) for the SCARA.

8.2.2 Preliminary examinations

Before starting the experiment, we collect some preliminary observations regarding the use case.

Controlled system

The SCARA is a fully controlled system. The goal of a well-designed controller is to follow a given reference as accurate as possible despite external disturbances. In this use case, we give motor positions for the three SCARA axes and expect them to be realized properly. However, the incorporation of a controller complicates the actual training task because it compensates for every intensity of the learned friction within

the physical possibilities of the electric motors. As a result, small perturbations in the learned friction model may not lead to changes in the TCP position, or the written text respectively, because the controller is able to cancel the disturbances out. This circumstance requires that we define the optimization objective *not* on the TCP position, but another quantity that is *directly* influenced by the learned friction model. This is further investigated in the next paragraph.

Mismatch between training objective and quantity of interest (QoI)
Often, the training goal is defined on the quantity to be optimized, the QoI. In the considered use case, we want to optimize the TCP position — or the written word correspondingly — however, we define the optimization objective on another quantity. For this use case, we select the motor currents, that increase for increasing friction, and which can easily be measured[4] in a real system. The expectation is that if the hybrid model makes good predictions for the motor currents, we will also find a good prediction for the TCP position — including the stick-slip oscillations.

8.2.3 Experiment

The introductory experiment of the SCARA differs from the two further experiments that we will conduct in Secs. 8.3 and 8.4. The focus of this experiment is not on validating the methods in industrial use, but rather on examining the usefulness of the approaches in greater depth using a more transparent example compared to the ones including restrictions w.r.t. data and/or the model. This is most clearly demonstrated in the fact that by generating data using a simulation model, we actually know which physical effect is to be learned *exactly* and how it relates to the physical system model. Of course, to such an extent, this is never the case in real-world applications. Nevertheless, it is worth noting that this example represents a complex system including many real-world challenges, and that is significantly more demanding than the average model found in publications (benchmark ODEs) in the field of SciML.

Hybrid model architecture
A hybrid model architecture for the SCARA can be derived step by step, based on a priori knowledge of the system to be modeled. Although we know exactly what the physical effect to be learned (stick-slip) looks like, we derive the architecture from the perspective of a development engineer who only assumes that the effect to be modeled is a friction-like effect.

Core architecture We assume, that the state derivatives approximated by our simulation model are relatively accurate, meaning the physical model prediction is worth reusing. However, we are not sure about how the effect interacts *mathematically*, so if it could be expressed by manipulating the existing derivatives or by addition of further terms. Therefore, we choose the architecture for manipulation of dynamics featuring gates (s. Sec. 7.2.5) as starting position, which consists of building blocks like pre- and post-processing (s. Sec. 4.5.1), gates (s. Sec. 4.5.2), and an FMU (s. Sec. 7.3) of the simulation model.

[4] Almost any motor controller is able to provide the motor currents.

ANN input The consideration of which variables to choose as input for the ANN corresponds to the consideration of which variables the effect to be modeled depends on. Because we assume a friction effect, we know that the normal force plays a crucial role. Further, most friction depends on the velocity, therefore the TCP velocity is also added as ANN input. The friction force acts directly on the TCP, however, the ANN calculates the accelerations for the robot axes, which we investigate in the next bullet point. As a result, friction must be investigated w.r.t. the rotational axes. Translated to the accelerations of the individual axes, we find that the friction torque depends on the axes positions. Therefore, we add the axes positions (states) as further inputs for the ANN.

ANN output Further, we know the physical states of the system, and especially that the mechanical subsystem is a second order ODE. Therefore, we apply a SOP structure (s. Sec. 3.4.6) for the mechanical subsystem, meaning only the angular accelerations of the robot joints are subject to modification and the angular velocities are forwarded. Finally, only the mechanical subsystem shall be manipulated by the ANN, because the goal is to learn for (mechanical) friction. This again results in forwarding the derivatives of the electrical currents straight to the ODE solver.

The resulting HUDA-ODE architecture can be investigated in Fig. 8.7 for the right-hand side $\boldsymbol{f}$, in Fig. 8.8a for the output function $\boldsymbol{g}$, and in Fig. 8.8b and Fig. 8.8c for the event functions $\boldsymbol{c}$ and $\boldsymbol{a}$. While the considerations above only impact the structure of the right-hand side of the hybrid model, the output and event functions are not affected and equal the corresponding FMU functions, so the hybrid model *inherits* these functions from the FMU, or the physical model respectively.

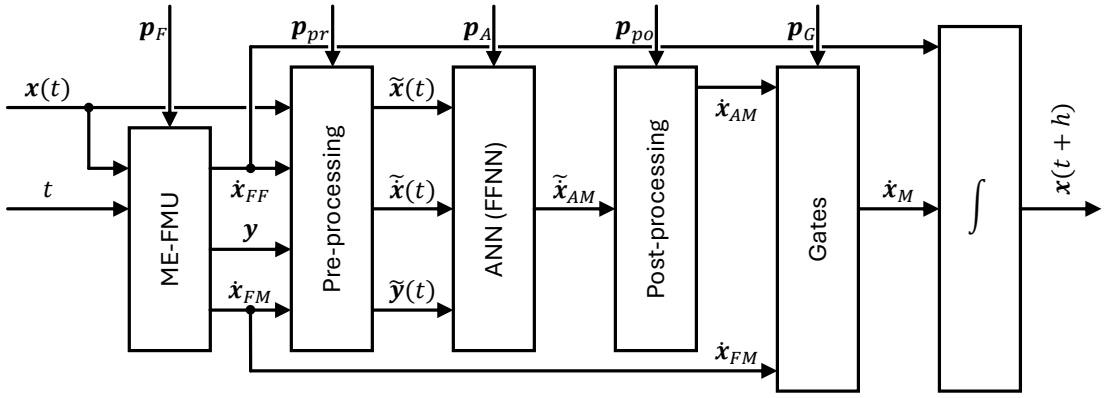
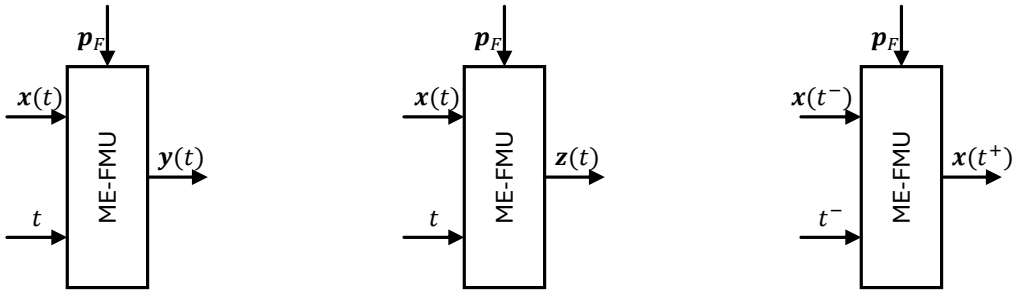


Figure 8.7: The function $\boldsymbol{f}$ (and an integrator) of the HUDA-ODE for the hybrid SCARA model. Figure adapted from Thummerer *et al.* [147] and extended. The time t and state $\boldsymbol{x}$ are passed on to the FMU, that computes the state derivative, separated in the part that is forwarded $\dot{\boldsymbol{x}}_{FF}$ and the part that is also calculated by the ANN $\dot{\boldsymbol{x}}_{FM}$. The state derivative (consisting of $\dot{\boldsymbol{x}}_{FF}$ and $\dot{\boldsymbol{x}}_{FM}$), the state $\boldsymbol{x}$, and further outputs $\boldsymbol{y}$ from the FMU are pre-processed and the FFNN calculates the derivative for the motor accelerations, which are post-processed to $\dot{\boldsymbol{x}}_{AM}$. The *gates* block determines the proportionate composition of $\dot{\boldsymbol{x}}_M$ from the ANN dynamics $\dot{\boldsymbol{x}}_{AM}$ and the FMU dynamics $\dot{\boldsymbol{x}}_{FM}$. The remaining state derivatives $\dot{\boldsymbol{x}}_{FF}$ are forwarded to the ODE solver and numerically integrated to obtain $\boldsymbol{x}(t+h)$. The physical parameters for the FMU $\boldsymbol{p}_F$ are assumed to be known, whereas the parameters for the ANN $\boldsymbol{p}_A$, the gates $\boldsymbol{p}_G$, and pre- $\boldsymbol{p}_{pr}$ and post-processing $\boldsymbol{p}_{po}$, are determined by optimization (training).



(a) The output function $\boldsymbol{g}$.

(b) The event condition function $\boldsymbol{c}$.

(c) The event affect function $\boldsymbol{a}$.

Figure 8.8: The HUDA-ODE functions $\boldsymbol{g}$, $\boldsymbol{c}$, and $\boldsymbol{a}$ for the SCARA.

Hyperparameter optimization

For all experiments, the resource within the individual runs is rescaled, so that the most expensive run within the entire optimization has a budget of $1,000$ s (instead of 81 s) of training data, independent of the chosen batch element duration. We train the proposed hybrid model for the ECM under the introduced hyperparameters in Sec. 8.1.3, no further hyperparameters are added.

Baselines
As introduced, a striking attribute of the SCARA use case is that we define our optimization objective on the electrical currents of the motors (because they are accessible easily by the motor controller), while the actual QoI is the position of the TCP. While this is possible within hybrid modeling, because the change in electrical current propagates through the physical model and influences the TCP position, this does not apply to a pure ML model. Note, that we could train a ML model to replicate the electrical currents[5], but not the TCP position because it is not part of the cost function. Changing the cost function to also include the TCP position would be a hard change to the use case itself and is therefore discarded. We therefore only validate against the original physical model (FMU) for the SCARA. Note, that for the other validation examples (ECM and PCS), the cost function is applicable to pure ML models as well, and we therefore include a corresponding ML baseline, of course.

8.2.4 Results & discussion

In the following, we investigate the results of the best hyperparameter run for the hybrid SCARA model. The hyperparameter optimization result can be investigated in Sec. A.6. We focus on a qualitative analysis of the results, and the quantitative summary can be found in Tab. 8.3. The datasets for training, validation, and testing are examined separately, and selected letters are investigated in more detail.

Training
We expect a successfully trained neural FMU to deliver significantly improved results over the original FMU on the known training dataset. We start by comparing the measurements of the electrical current over the two electric motors for both motor axes, see Fig. 8.9.

We find significantly larger currents within the data measurements, compared to the FMU prediction. This is because of the not modeled friction, in fact, the motor torques only need to overcome the inertia of the multi-body system. Especially during the last seconds of the simulation, where the influence of the stick-slip is the most, the motor currents are the largest. The neural FMU provides a very close fit compared to data, which further mirrors in a small AE. We find that the FMU, on the other hand, predicts (too) small currents along the entire simulation. Especially interesting is that even for the letters on the left side of the paper plane, where the physical model is able to make good predictions, there are large deviations for the predicted current between the FMU and data. This shows the impact of the controller, that is able to match the TCP target trajectory independent of the missing friction (FMU) or learned friction (neural FMU) — however, only to a limited extend: For the right-most letters, the linear controller reaches its limitations. This motivates to further investigate the predictions of the TCP position. Investigating the part of the trajectory of the TCP position, where the normal force at the pen tip is larger than zero, we find the resulting drawing, see Fig. 8.10.

[5]However, because we are using only a single training trajectory, the ML model will most likely overfit.

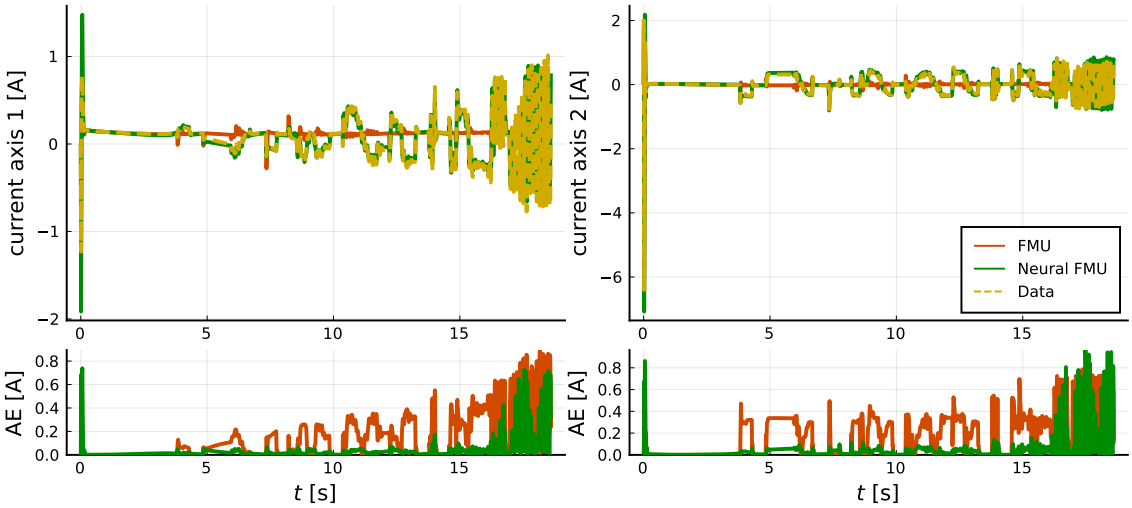


Figure 8.9: Results for the SCARA experiment, comparing FMU and neural FMU predictions for the electrical current while writing the word "train". The upper plots show the electrical currents, the lower plots the absolute deviation compared to GT data.

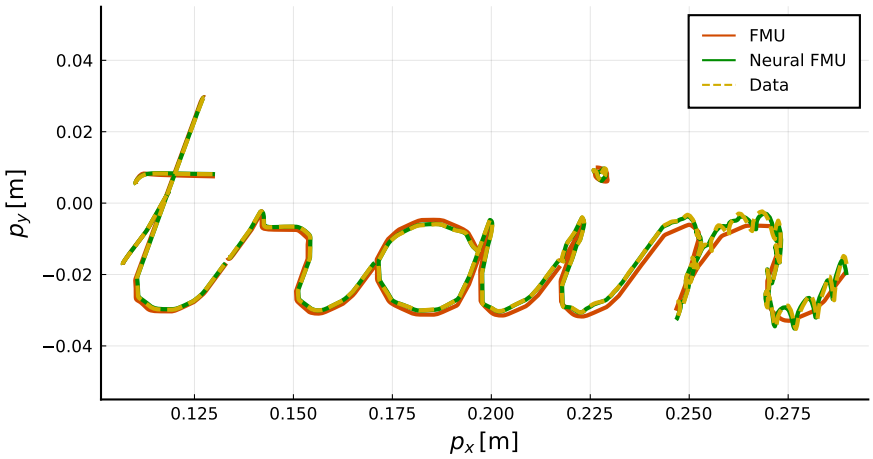
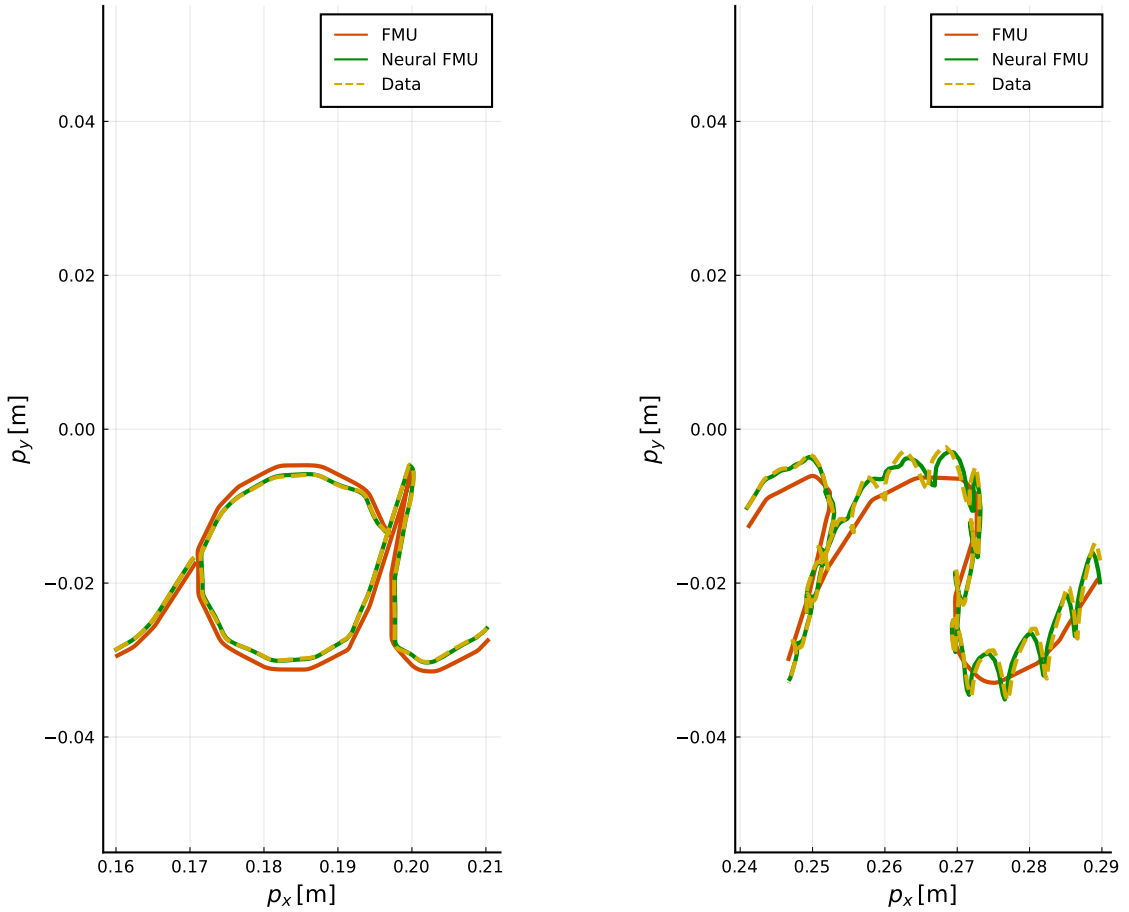


Figure 8.10: Results for the SCARA experiment, comparing FMU and neural FMU prediction for the written word "train".

As to expect, the neural FMU nicely captures the oscillation by the stick-slip friction, as it is especially placative for the letters on the right side of the paper. We proceed investigating the presence and non-presence of the stick-slip for the two letters "a" (left side of paper, little to no stick-slip) and "n" (right side of paper, much stick-slip), see Fig. 8.11.



(a) The letter “a”.

(b) The letter “n”.

Figure 8.11: Results for the SCARA experiment, comparing FMU and neural FMU prediction for the written letters “a” and “n” in the word “train”.

While no characteristic stick-slip-effects are apparent for letter “a”, we find a small deviation between the physical model and the data, which can be attributed to unmodeled viscous friction that does not need to be compensated for by the controller. We see that the neural FMU is able to close this small gap, and we suspect that the hybrid model therefore was able to learn a good friction model for the range of simple sliding friction. This assumption can be further supported by investigating Fig. 8.12, which shows that the predictions of the electrical currents by the neural FMU are also very well represented. The FMU, on the other hand, constantly shows a too low current level.

For the letter “n”, on the other hand, we find a very pronounced stick-slip effect in Fig. 8.11b. While the physical model (FMU) is only capable of approximating an average value instead of the oscillation, the hybrid model (neural FMU) is capable of predicting the amplitude and frequency of the oscillation accurately. This again can be deeper examined by investigating the electrical currents in Fig. 8.13. Here, the neural FMU shows for the electrical current at axis 2 a greater maximum — however, a much smaller average — deviation, compared to the FMU. This is because of a phase shift in the oscillation, leading to large values for the MAE. The shift is most likely caused by a deviation earlier in the simulation — note, that 16 s of robot movement are simulated before the letter “n” is started to be drawn. However, the

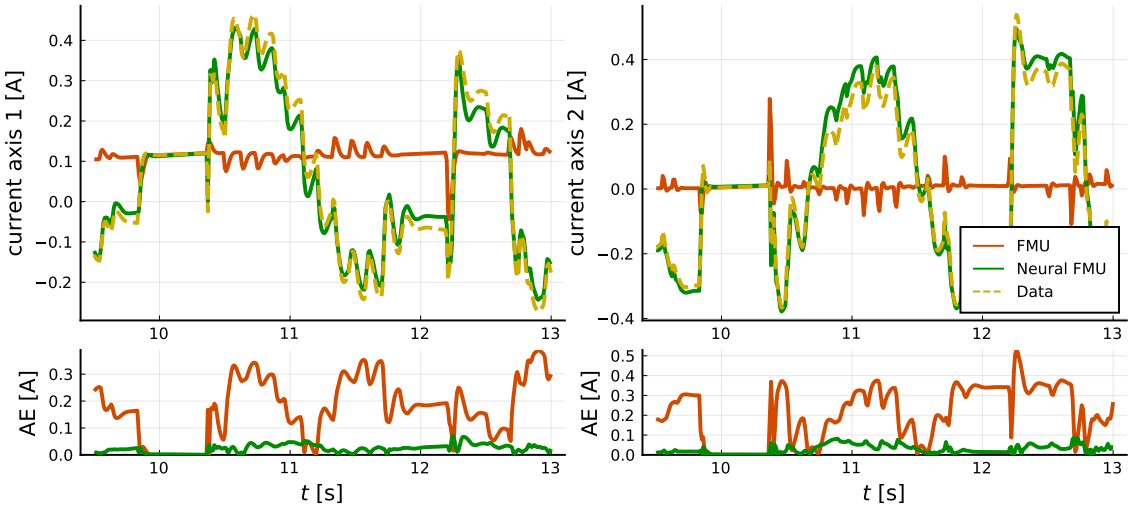


Figure 8.12: Results for the SCARA experiment, comparing FMU and neural FMU predictions for the electrical current while writing the letter "a" in the word "train". The upper plots show the electrical currents, the lower plots the absolute deviation compared to GT data.

deviation for the electrical current gives a disproportionately bad impression. Note, that the amplitude, phase, and frequency of the oscillation on position level are captured very well as we found in Fig. 8.11. Error metrics with a stronger focus on comparing oscillating signals, such as *dynamic time warping* [148], [149], [150], will lead to significantly smaller errors.

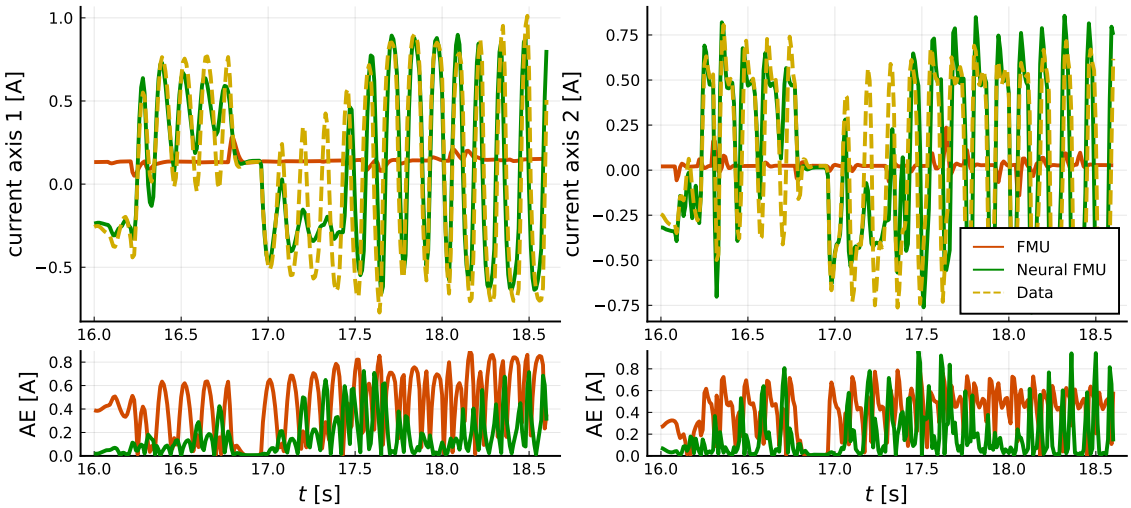


Figure 8.13: Results for the SCARA experiment, comparing FMU and neural FMU predictions for the electrical current while writing the letter "n" in the word "train". The upper plots show the electrical currents, the lower plots the absolute deviation compared to GT data.

Even though these results look very promising overall, further testing on unknown data is needed to verify whether the model has also learned to generalize. For this use case, we therefore continue to check the model performance for validation data, as this is the more interesting dataset (contains more variation in letters), and refer to the appendix for test data results (s. Sec. A.9).

Validation
The validation data is not part of the training data, but was used to select promising hyperparameter runs. Note, that for proper *validation of the hybrid model*, test data must also be checked. At this point, however, we actively decide to investigate validation instead of test data, because the involved letters are more interesting to discuss. Interested readers are referred to Sec. A.9 for the investigation of test data. In the following, we examine the model quality on new and known letters, and further note that the letters in the validation dataset are smaller than those in the training dataset, see Fig. 8.14.

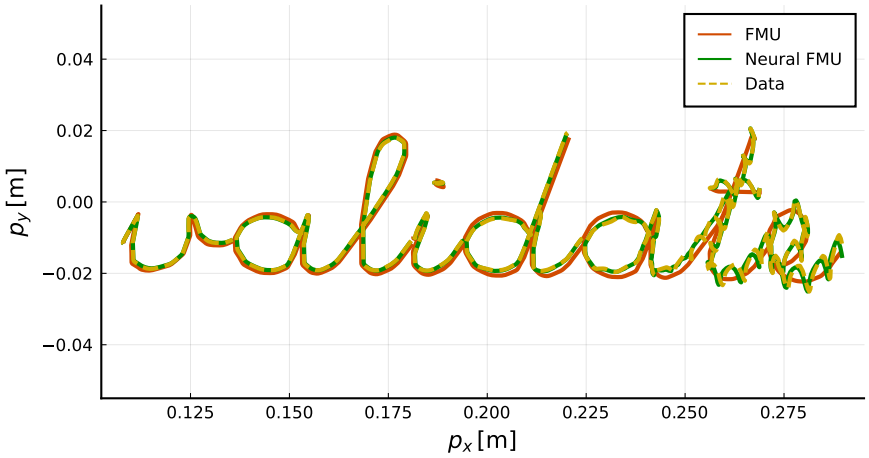
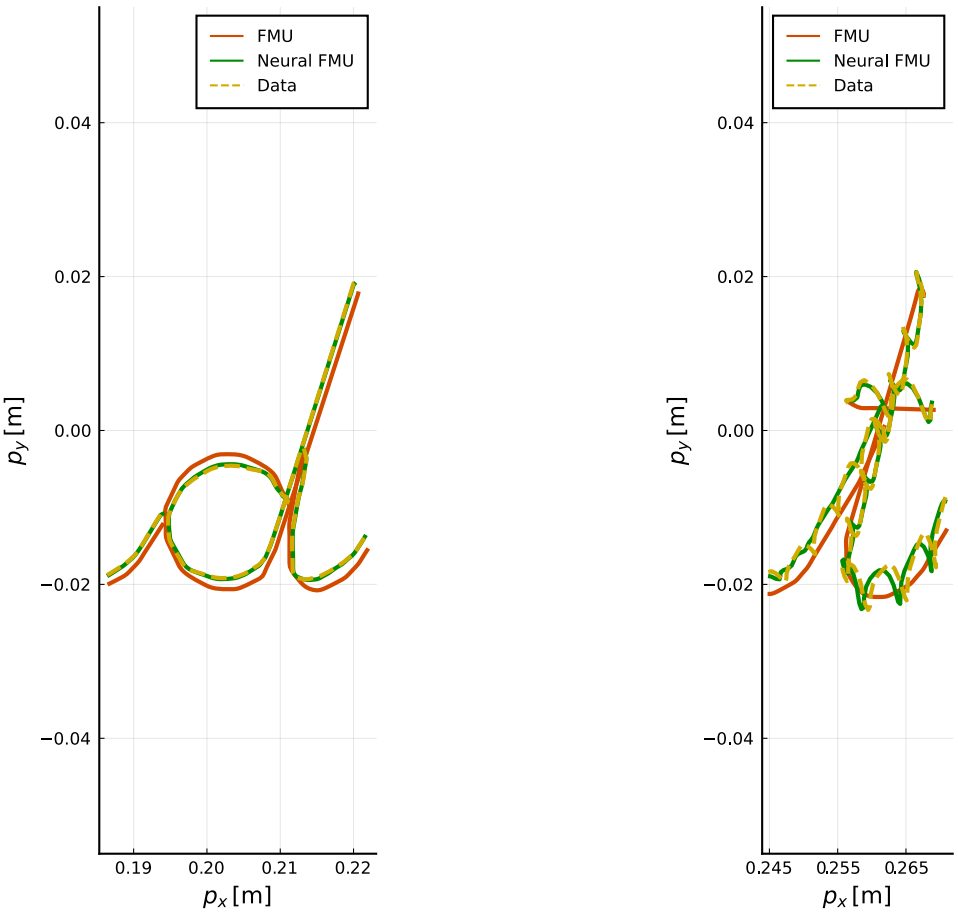


Figure 8.14: Results for the SCARA experiment, comparing FMU and neural FMU performance on the word "validate".

On a positive note, the qualitative picture that emerges is very similar to that in the training data run, which suggests that the model generalizes well. We find a good fit of the hybrid model for both the left side (with little stick-slip) and the right side (with increasing stick-slip). We continue by examining two letters in particular: "d", a new letter, but similar to the known "a", and "t", a familiar letter, but which, compared to the training data, is further to the right and therefore affected by stick-slip, see Fig. 8.15.

Similar to the word "train", we find a nice fit for the letter, that is not affected by stick-slip, the letter "d" in this case. The almost constant deviation of the FMU is not present in the neural FMU prediction. This mirrors in a good fit of the electrical currents as well, compare Fig. 8.16.



(a) The letter “d”.

(b) The letter “t”.

Figure 8.15: Results for the SCARA experiment, comparing FMU and neural FMU performance on the letters “d” and “t” in the word “validate”.

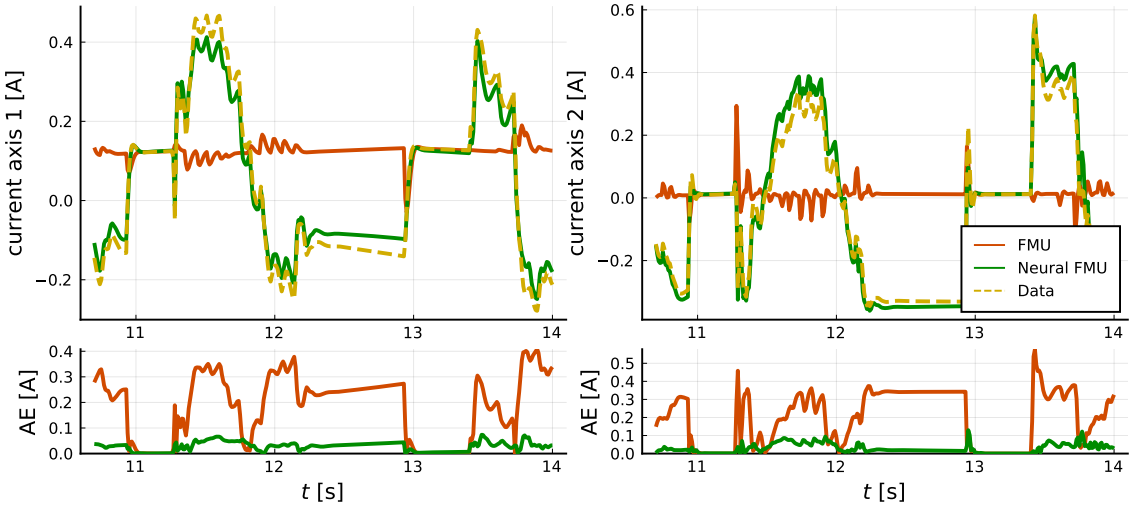


Figure 8.16: Results for the SCARA experiment, comparing FMU and neural FMU predictions for the electrical current while writing the letter “d” in the word “validate”. The upper plots show the electrical currents, the lower plots the absolute deviation compared to GT data.

For the letter "t" in Fig. 8.15b, which is now affected by stick-slip, we find a significant improvement compared to the pure FMU. The characteristic oscillation is captured very well. There are only very small deviations in amplitude, frequency and phase. We close the result presentation and discussion by investigating the electrical currents for the letter "t" in Fig. 8.17.

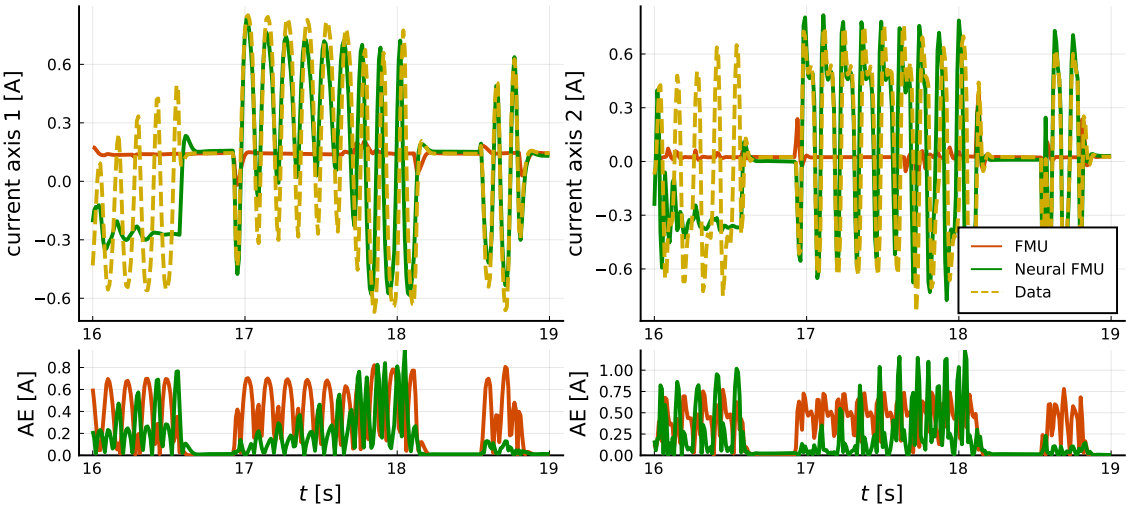


Figure 8.17: Results for the SCARA experiment, comparing FMU and neural FMU predictions for the electrical current while writing the letter "t" in the word "validate". The upper plots show the electrical currents, the lower plots the absolute deviation compared to GT data.

Here, we see an improvement between FMU and neural FMU, however, we can also see the source for the yet not perfectly predicted letter "t", which is the neglection of the electrical current oscillation at the very beginning of the letter, where the middle line is drawn from bottom to top. However, we need to recall that the letter "t" was not affected by stick-slip in the training dataset and is in general one of the most complex and challenging letters to draw.

Summary

We can summarize that hybrid modeling can be applied to compensate for unmodeled physical effects, even if we provide interfaces that do not directly match the involved quantities. For example, while the not modeled stick-slip effects depends on the TCP velocity and acts as additional translational force on the TCP, the hybrid model is able to learn a representation of the stick-slip that directly acts on the axes accelerations. This impressively shows an additional benefit of applied SciML, because it is not required to provide an *ideal* architecture — in the sense that learning a minimal symbolic representation of the physical effect is possible — but only an architecture that allows accessing all relevant information, even if only indirectly. Doing so, we can take into account the requirements of a genuine hardware setup — like a SCARA demonstrator — in which some quantities (like the electrical current) might be easier to measure than others (like the TCP position). For the SCARA, we were able to improve the hybrid model by multiple factors, depending on the investigated letters and quantities, see Tab. 8.3.

Table 8.3: Comparison of the electrical currents and Euclidean distance of the TCP for the FMU and neural FMU on different words and letters. The sub-word, for which the errors are calculated is given in square brackets.

Model	Word	cur. 1 [A] MAE ↓	cur. 2 [A] MAE ↓	pos. [mm] Euclid. dist. ↓
FMU (physical)	[train]	0.1464	0.1816	0.76
Neural FMU (hybrid)	[train]	0.041	0.0482	0.19
FMU (physical)	tr[a]in	0.1736	0.2077	0.85
Neural FMU (hybrid)	tr[a]in	0.0235	0.0284	0.1
FMU (physical)	trai[n]	0.4286	0.4036	2.16
Neural FMU (hybrid)	trai[n]	0.1651	0.1817	0.76
FMU (physical)	[validate]	0.1767	0.2362	1.03
Neural FMU (hybrid)	[validate]	0.0678	0.0974	0.45
FMU (physical)	vali[d]ate	0.1856	0.1973	0.99
Neural FMU (hybrid)	vali[d]ate	0.0282	0.0284	0.13
FMU (physical)	valida[t]e	0.2897	0.3097	1.46
Neural FMU (hybrid)	valida[t]e	0.159	0.1917	0.74
FMU (physical)	[test]	0.1765	0.2285	1.06
Neural FMU (hybrid)	[test]	0.0985	0.1272	0.63
FMU (physical)	t[e]st	0.1341	0.1618	0.6
Neural FMU (hybrid)	t[e]st	0.0153	0.0225	0.07
FMU (physical)	tes[t]	0.3215	0.2812	1.59
Neural FMU (hybrid)	tes[t]	0.2013	0.2168	0.94

8.3 Automotive engineering: Electronically commutated motor (ECM)

Within this section, the validation use case for the electronically commutated motor (ECM) is introduced and processed. This section is based on an own parallel publication [147], but features more detailed examinations. For example, different scheduling techniques are investigated and a deeper discussion of results is provided.

8.3.1 Introduction

This use case is from *Robert Bosch GmbH* and was worked on during the *UPSIM* project. For reasons of IP, the physical model can not be shared in detail within this thesis. What at first glance appears to be a restriction and casts doubt on its suitability as an example for validation, is on the other hand a good example of the *real* process in industry: We are often forced to work with models, that we do not fully understand or have limited access to. In engineering practice, it must be sufficient to know the purpose of the model, the underlying assumptions and the interfaces to other models. Being able to investigate and influence every part of any model is unrealistic in real engineering scenarios.

Use case
The ECM model, that is investigated within this use case, is part of the Bosch integrated power brake (IPB) system. The structure of the system is visualized simplified in Fig. 8.18. The IPB consists of three major parts: The ECM, a suitable

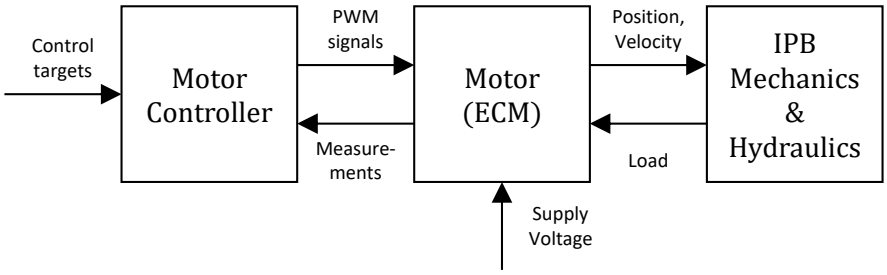


Figure 8.18: A rough overview of the IPB simulation model, consisting of the motor and motor controller, as well as additional mechanical and hydraulic subsystems. Figure adapted from Thummerer *et al.* [147].

motor controller and the mechanical / hydraulic subsystem that is actuated. The motor and motor controller operate together as *control loop*. After modeling the IPB simulation, the resulting model needs validation. Based on experiments in the field, it was discovered that the ECM part of the model needs enhancements in terms of prediction accuracy to allow for special SiL testing. Only limited data from measurements of a real demonstrator is available. This situation — having a simulation model available, together with some (but limited) data — motivates the use of *hybrid modeling* to increase the simulation accuracy of the ECM model.

Physical model
The ECM is not available as white-box model, but only as compiled FMU, together with a brief model specification. The model of the ECM is based on physical equations but features some application-specific extensions. However, most relevant for investigation of the use case are the interfaces of the model, so the inputs and outputs. The ECM computes the rotor angle and velocity (outputs) based on the phase currents, supply voltage and mechanical load (inputs). Further, the model features four states, two for the mechanical subsystem (position and velocity) and two for the electrical subsystem. Finally, the model further contains an unknown number of discrete states, that are hidden by the FMI.

Data
For this experiment, three measurements of the real system are used, one for training, validation, and testing each. Each measurement corresponds to a single, short driving maneuver between 10 s and 20 s. Data includes measurements of the current over the motor phases (U, V and W), the motor supply voltage, mechanical load caused by the hydraulic subsystem, and the motor rotor speed as well as the rotor position (limited to the interval between $-\pi$ and π). Data was recorded via the control unit bus and is subject to typical limitations of real data such as noise, unknown external influences, and non-equidistant measurement points.

8.3.2 Preliminary examinations

We make some preliminary examinations before starting with the actual validation experiment, to highlight the special boundary conditions of this application in the field.

Accessibility
Because of IP, none of the three subsystems is available in symbolic form (equations). Furthermore, only the ECM is shared as ME-FMU, together with a model description of inputs and outputs, but explicitly no access to the equations within the model. For the other subsystems, only data at the interfaces to the ECM is shared. At this point it is important to highlight that such restrictions are typical for working in industry, for example, if simulation models are shared between car manufacturers and original equipment manufacturers (OEMs).

Solver
The model is solved with *Heun's method* [151], a two-stage explicit Runge-Kutta method. This is a given requirement for the later operation of the model. In the final application, a fixed solver step size of 10^{-4} s is required, however for training, we use step-size control and only limit the maximum step size to 10^{-4} s. Even if the fixed step size is suitable to solve the physical model with sufficient accuracy, during training the dynamic step size allows for handling of stiffness/instability within the hybrid model, that can occur during certain phases of training.

Control loop
The system contains a control loop. The ECM can be controlled in different modes, to reach a target position, speed or current. In the considered example we only focus on the position control. Because a software controller not only compensates the control deviation, but also *modeling errors*, it is crucial to break the control loop for training the hybrid model for the ECM in order to separate the modeling error and the control deviation. For this purpose, measurements are taken from the closed loop system, but training of the ECM happens open-loop.

Note, that the SCARA contains a control loop for every actuated axis as well, however, we did not break the control loops in this case. Therefore, we briefly examine the difference below. Whereas for both use cases the ANN learns for the angular acceleration of a controlled electric motor, the cost functions are defined on different QoIs: For the SCARA, the loss function is defined on the electrical current, that is *algebraically* connected to the motor torque and therefore the rotor acceleration. This results in the fact that changes to the acceleration *directly* influence the QoI — the motor current — and therefore the loss value. In this case, the controller has no chance to interact because the control input — the motor position — does not change immediately, but only *after* the next integration step. As a result, the optimizer recognizes the change in the QoI *before* the motor controller, and can therefore perform proper optimization of the ANN parameters. The ECM, on the other hand, uses a loss function defined on the rotor position and velocity, so QoIs that are not algebraically connected to the acceleration, but via *integration*. In this case, the motor controller as well as the optimizer recognize changes to the QoI after the integrator step simultaneously, what must be prohibited for proper optimization.

Shift in data
During preliminary analysis of the measurement data, we found small, but noticeable temporal deviations between the simulation model of the ECM and the real system, compare Fig. 8.19. It is unclear, if these deviations result from misaligned measurements or missing dynamical effects in the simulation model. This provides motivation to take measures for a possible correction.

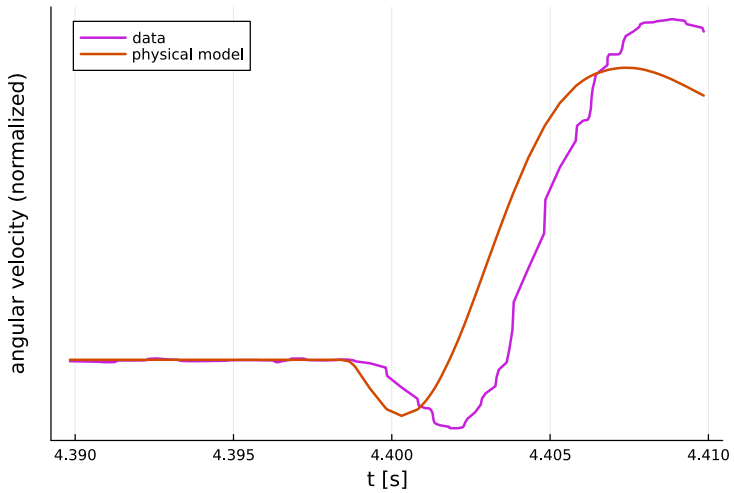


Figure 8.19: The shift ($\approx 2\,\text{ms}$) between data and physical model for the ECM angular velocity. Figure adapted from Thummerer *et al.* [147].

Low variance in data
Further investigating the data, we find that large parts of the measurements are very similar, for example segments where the motor stands still or is operated with constant speed and load, compare Fig. 8.20. Looking at the data as a whole, the largest part consists of such "boring" measurements, and only some "interesting" moments are included, e.g. driving the motor to a different position quickly. Applying common training strategies, like mini-batching with random scheduling, to such data with large parts of low variance causes a strong bias towards the non-dynamic parts of data. This is not intended, or even the opposite is the case: We want high-dynamic parts to be captured by the hybrid model or ideally, a healthy balance across the dynamic spectrum. This requires additional steps.

Unknown discrete state
The ECM model contains an unknown number of discrete states. Note that although the discrete state is not required for proper simulation (if initialized correctly), it does pose a problem if batching is applied. In FMI, the discrete state is determined by solving a dedicated initialization problem. Many simulations (like the ECM here) start in a non-dynamic state (so $\dot{\boldsymbol{x}}(t_0) \approx \mathbf{0}$) where the discrete state can be initialized in a traceable way, even if not known. This does not apply if we start the simulation in (possibly) high dynamic locations in state space — as we inevitably do during equidistant batching. For this purpose, we apply a *dummy discrete state* (s. Sec. 7.3.1) for the ECM use case in order to apply proper mini-batching.

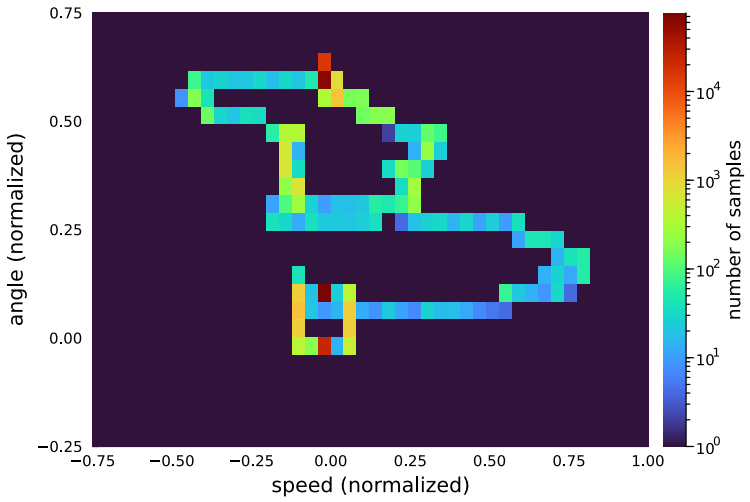


Figure 8.20: Very heterogenus data availability for training, visualized for ECM speed and angle. Large parts of the dynamic trajectory (speed $\neq 0\,\mathrm{m/s}$) in the mechanical state space offers only $10 - 100$ data samples (blue, turquoise), whereas non-dynamic parts (speed $= 0\,\mathrm{m/s}$) is represented with up to 10^5 samples (dark red).

8.3.3 Experiment

In the following, the validation experiment is set up: We derive a hybrid model architecture, briefly set up a proper hyperparameter optimization and introduce the baselines to compare with. The former two steps directly correspond to the chapters of this thesis: Creation (s. Cha. 3), Initialization (s. Cha. 4), Training (s. Cha. 5), and Real World (s. Cha. 7). This experiment is performed similarly, but with reduced hyperparameter space, within the own work Thummerer *et al.* [147].

Hybrid model architecture

Based on the architecture for manipulation of dynamics featuring gates (s. Sec. 7.2.5), a suitable architecture is derived. Only a single state derivative, the rotor acceleration, is manipulated by the FFNN. The rotor velocity and the derivatives of the electrical subsystem are directly forwarded to the ODE solver. This corresponds to a SOP structure (s. Sec. 3.4.6) for the mechanical subsystem. The architecture is further extended by a differentiable block that is able to shift the input signal to compensate for static time offsets in data and achieve a better fit. The final architecture for the HUDA-ODE function $\boldsymbol{f}$, further consisting of introduced building blocks like pre- and post-processing (s. Sec. 4.5.1), gates (s. Sec. 4.5.2) and an FMU (s. Sec. 7.3), can be investigated in Fig. 8.21. As for the SCARA experiment, the output function $\boldsymbol{g}$, as well as the event functions $\boldsymbol{c}$ and $\boldsymbol{a}$, are inherited from the FMU (compare Fig. 8.8), with the modification that the delay correction block is applied here as well to obtain a shifted input signal.

Hyperparameter optimization

For all experiments, the resource within the individual runs is rescaled, so that the most expensive run within the entire optimization has a budget of 100 s (instead of 81 s) of training data, independent of the chosen batch element duration. We train the proposed hybrid model under the hyperparameters as introduced in Sec. 8.1.3,

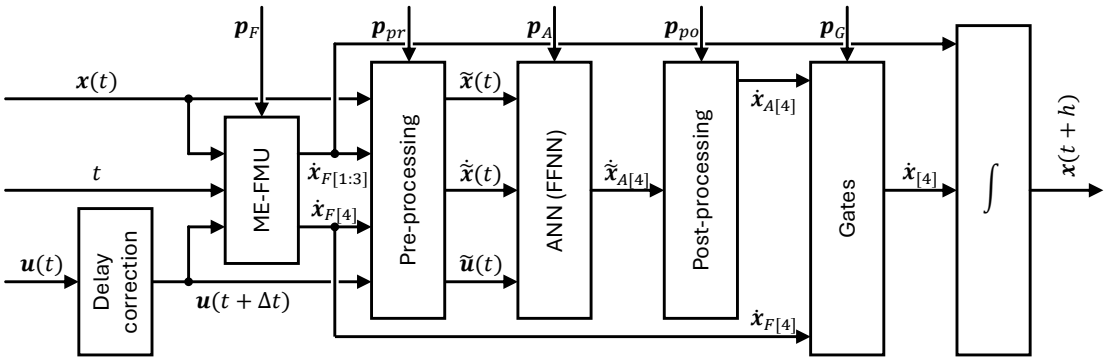


Figure 8.21: The function $\boldsymbol{f}$ (and an integrator) of the HUDA-ODE for the hybrid ECM model. Figure adapted from Thummerer *et al.* [147] and extended. The input $\boldsymbol{u}$ is shifted by the *delay correction* block. The shifted inputs $\boldsymbol{u}(t+\Delta t)$, time t and state $\boldsymbol{x}$ are passed on to the FMU, that computes a state derivative $\dot{\boldsymbol{x}}_F$. The original state $\boldsymbol{x}$, the shifted input $\boldsymbol{u}(t+\Delta t)$ and the state derivative from the FMU $\dot{\boldsymbol{x}}_F$ are pre-processed and the FFNN calculates the derivative for the motor acceleration, that are post-processed to $\dot{x}_{A[4]}$. The *gates* block determines the proportionate composition of $\dot{x}_{[4]}$ from the ANN dynamics $\dot{x}_{A[4]}$ and the FMU dynamics $\dot{x}_{F[4]}$. The remaining state derivatives $\dot{\boldsymbol{x}}_{F[1:3]}$ are forwarded together with $\dot{x}_{[4]}$ to the ODE solver and numerically integrated to obtain $\boldsymbol{x}(t+h)$. The physical parameters for the FMU $\boldsymbol{p}_F$ are assumed to be known, whereas the parameters for the ANN $\boldsymbol{p}_A$, the gates $\boldsymbol{p}_G$, and pre- $\boldsymbol{p}_{pr}$ and post-processing $\boldsymbol{p}_{po}$, are determined by optimization (training).

however, extend the hyperparameter space by the following two parameters:

Loss scaling The loss contains both mechanical states, the ECM rotor speed and the rotor position. A varying scaling between 0 % and 100 % is applied to the speed, with increments of 10 %. The scaling of the angle is chosen opposite. This way, different ratios, but also training on speed or angle only, are investigated during the hyperparameter optimization.

Scheduling strategy Multiple scheduling strategies are compared. This covers random scheduling, loss accumulations scheduling and auto-encoder-based latent-space-sampling (see Sec. A.4). This measure is applied to handle the significant data heterogeneity in the measurements, as previously introduced.

Baselines

In the sense of hybrid modeling, we use the two extremes of modeling approaches as baseline for performance evaluation: The physical model of the ECM, based on physical equations and compiled as FMU, and a neural ODE as representative for ML without any physics-based a priori knowledge. We deliberately select the neural ODE as representative for ML because we assume that this model type is one of the most powerful for this task. For example, it was shown that neural ODEs are able to deliver up to an order of magnitude more accurate results when learning for various nonlinear physical system, compared to *classical* ML approaches like linear and neural state-space models [152]. Further, it was found that they were less sensitive w.r.t. hyperparameters [152].

Whereas the application of the physical model as baseline is straightforward, additional considerations are necessary for the ML model. For reasons of fairness, we perform the hyperparameter optimization for the neural ODE again, and extend the hyperparameter space so that it suits the different model type. For example, the investigated learning rates are up to 100 times larger compared to the very sensitive hybrid model. Nevertheless, in this experiment we want to keep the general boundary conditions the same for all three models (physical, hybrid and ML) in order to enable comparability in applications with very limited data. Note, that the number of training steps and the size of the ANN are very small within the hybrid model, for conditions as we know them from ML. Another important point is that without prior knowledge of physics, the system dynamics in the entire operation state space of the model must be learned. Only small parts of the operation state space are captured in the training data, so pure ML has by design no real chance of representing a well generalizing model. We therefore assume that the neural ODE will have no chance of achieving similar results compared to the hybrid model under *identical* training conditions, which will underline the superiority of hybrid modeling in such applications with extremely little data. Nevertheless, it is good scientific practice to also include these results in quantifiable terms, which we do in the following.

8.3.4 Results & discussion

In the following, we investigate the results of the best hyperparameter run (see Sec. A.7) for the hybrid ECM model. We investigate how the model compares to the pure physical and pure ML model, and will show, that hybrid modeling is a valuable measure for use cases with limited access to data.

Time delay
The learned time offset from data is $2.33 \cdot 10^{-5}$ s. Although this value slightly corrects the observed temporal offset in the data, the value is smaller than expected. If we recall Fig. 8.19, we assume a necessary correction in the range of $\approx 2 \cdot 10^{-3}$ s. We deduce from this observation that the temporal offset cannot be corrected statically via a temporal shift of the input signals — which speaks for the correctness of the physical modeling — but can actually be compensated via a correction of the dynamics (thinkable e.g. via learning a more complex damping behavior).

Training
Even if this does not yet allow us to assess the generalizability of the hybrid model, we start by evaluating the training results. The training simulation trajectories for the different investigated models is given in Fig. 8.22, which visualizes the motor rotor angle over time.

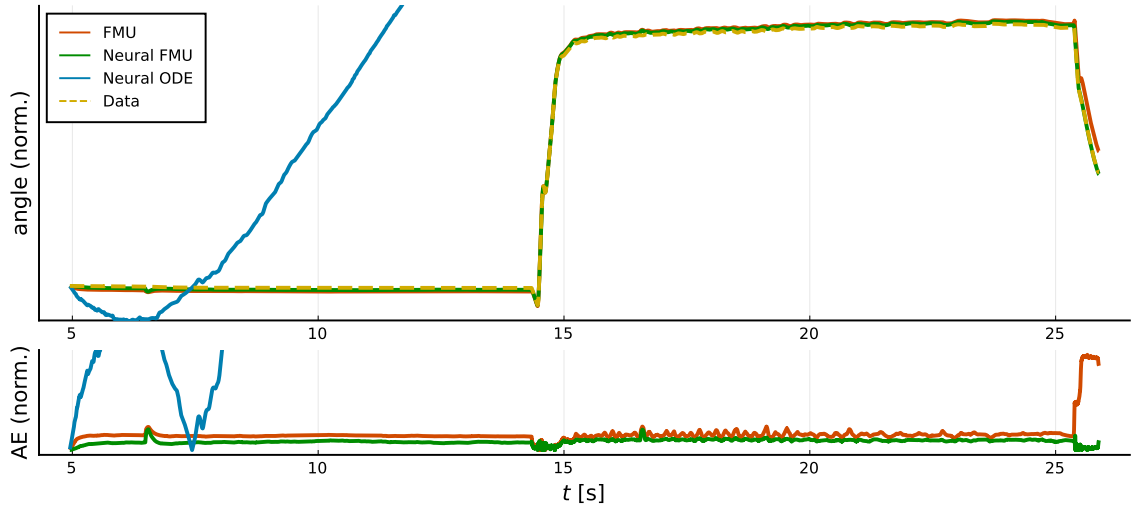


Figure 8.22: The ECM rotor angle on the training scenario, comparing the physical (FMU), hybrid (neural FMU), and ML (neural ODE) model. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

By investigation of the rotor angle simulation results, we find that the ML model (neural ODE) trajectory provides only a very unsatisfactory fit. Despite the individual hyperparameter optimization, the neural ODE is not able to deliver a comparable result. This is due to the tiny amount of data, which is simply not sufficient to identify the right-hand side of the system based purely on data without prior knowledge. We will therefore ignore the neural ODE in the following and, for example, scale the plot axes without taking this curve into account, which leads to truncation of the neural ODE values in favor of the readability of the other trajectories.

The physical (FMU) and hybrid model (neural FMU) trajectories, on the other hand, lay close to each other. However, especially during the drop at the end of the curve, we find larger deviations for the physical model. In order to better assess the actual error, the deviation can be shown in isolation (lower plot). Here we find the expected large deviation at the end of the trajectory, but also that the hybrid model consistently shows a smaller deviation over time, compared to the physical model. In Fig. 8.23, we examine the last part of the trajectory in detail.

Here, we can clearly see that the neural FMU provides an excellent fit and lays almost exactly on the data trajectory, while the FMU features a varying offset. Indeed, we can observe that the FMU trajectory increases instead of decreases a little before $\approx 25.4\,\mathrm{s}$, indicating that the motor model turns in the wrong direction at this point, leading to the deviation. This is a common issue with ECMs in general, they are very sensitive w.r.t. the PWM control signals, and minor shifts can lead to running the motor out of sync — which is most probably the reason for the observed deviation.

Since *large* deviations at the position level can occur due to *small* errors at speed level (accumulation by integration), it makes sense to subsequently also examine the rotor speed, compare Fig. 8.24.

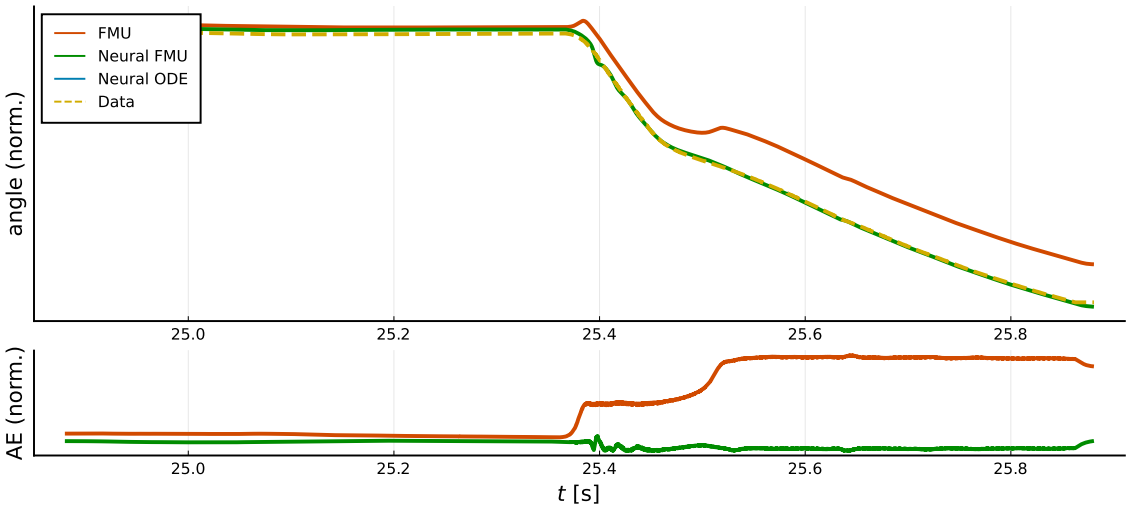


Figure 8.23: The ECM rotor angle on the training scenario, zoomed in for $t \in [24.9\,\mathrm{s}, 25.9\,\mathrm{s}]$. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

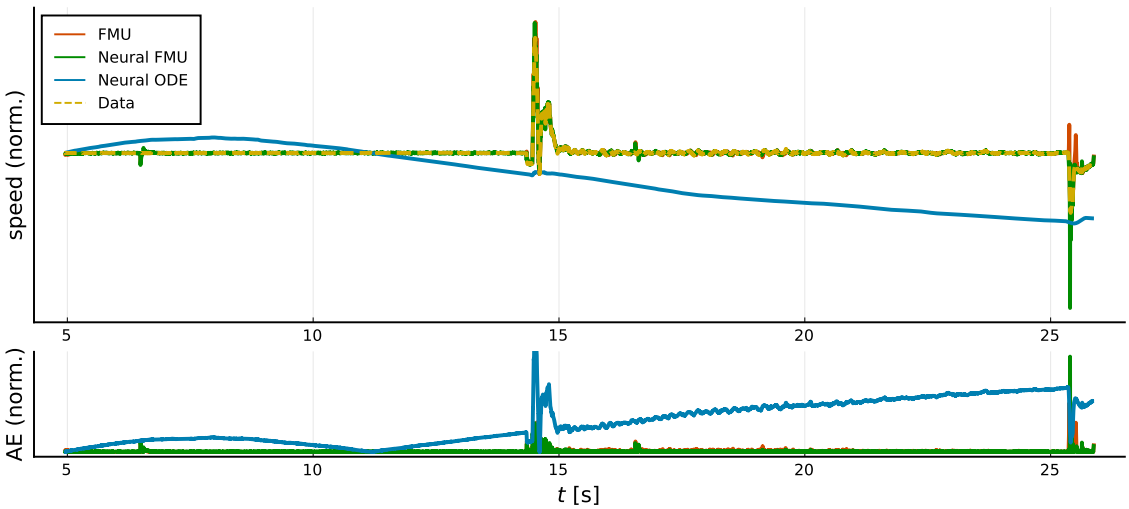


Figure 8.24: The ECM rotor speed on the training scenario, comparing the physical (FMU), hybrid (neural FMU), and ML (neural ODE) model. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

While the relevant simulation trajectories — so except the neural ODE — for the motor speed are again very close to each other, the investigation of the AE again provides more meaningful insights. What is most striking is that the difference in the error between the hybrid and physical models is not as great as for the motor position — what we expected from the differentiated quantity — and that in particular the hybrid model does not perform better at all times. This shows that the way in which the hybrid model manipulates its physical origin does not necessarily result in an improvement for every point in the state space. Due to the design of the cost function, which only rates the *average* improvement, the model is modified to generate better results *on average*. Nevertheless, we find significant improvements of the hybrid simulation model in specific areas, see Fig. 8.25.

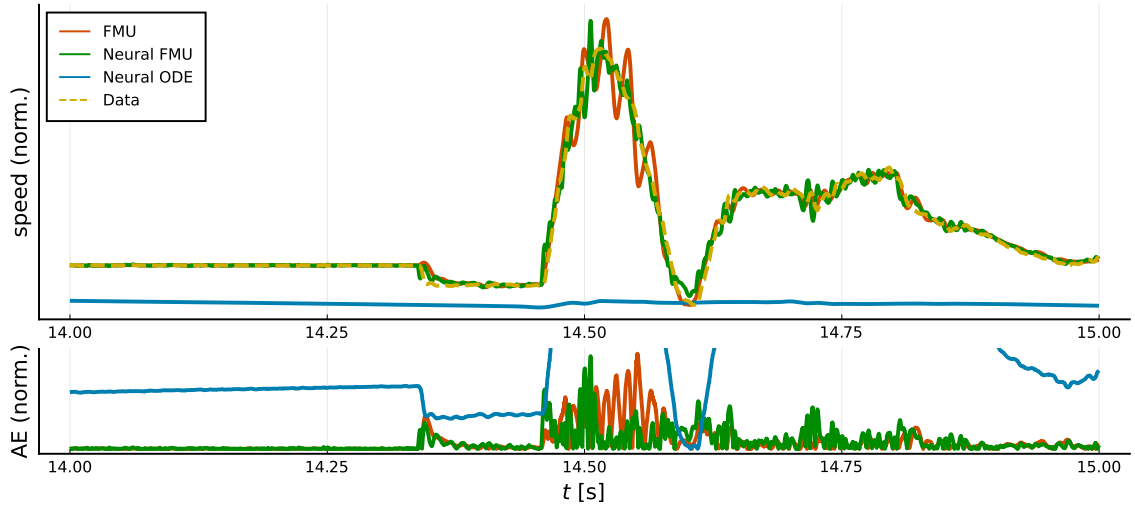


Figure 8.25: The ECM rotor speed on the training scenario, zoomed in for $t \in [14.0\,\mathrm{s}, 15.0\,\mathrm{s}]$. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

Here, we find an unintended oscillation around the data trajectory for the FMU, whereas the neural FMU does not show such a behavior, or only to a very limited extent. Finally, the overall improvement in the average can be verified by a quantitative evaluation of Tab. 8.4.

Table 8.4: Comparison of the solver steps (*stiffness*) and the MAE for the different models on training data. Errors are normalized for reasons of IP.

Model	MAE (angle)	MAE (speed)	Acc. steps	Rej. steps
FMU (physical)	0.0076	0.0046	471,575	54,094
Neural FMU (hybrid)	0.0041	0.0032	226,117	6,396
Neural ODE (ML)	0.6573	0.2213	253	20

To summarize the training outcome, we find that the hybrid model (neural FMU) makes better predictions regarding the rotor angle (factor ≈ 1.85) as well as the rotor speed (factor ≈ 1.39) in the training scenario, compared to the physical model

(FMU). This improvement is greater than it appears at first glance, as the training trajectory consists mainly of non-dynamic areas, which greatly average the improvements in the few dynamic areas. Therefore, we will examine a testing trajectory below that contains proportionally more dynamics. Further, we find that the numerical properties significantly improved, the hybrid model features only around half the number of required solver steps, and features around 8.5 times less rejected steps. The ML model (neural ODE) is not able to converge to a comparable minimum, because of the insufficient amount of data and the resulting limited investigation of the state space.

Testing

The investigation of the model performance on training data does not allow a statement whether the hybrid model is able to generalize well. Conventionally, we check the generalizability against a test dataset. The results on validation data are unremarkable (see Sec. A.10), so the results on test data are examined below. However, comparing the plots for the rotor angle on testing (s. Fig. 8.26) and validation (s. Fig. A.5) data, we notice that there is a certain similarity between the two maneuvers. Accordingly, it is important to explain that both braking maneuvers are more different than it appears at first glance, for example the maximum speed of the motor rotor differs by $> 11\,\%$ and the maximum external load by $> 8\,\%$. However, this difference is masked by the normalization. We proceed by investigating results for the motor rotor in Fig. 8.26.

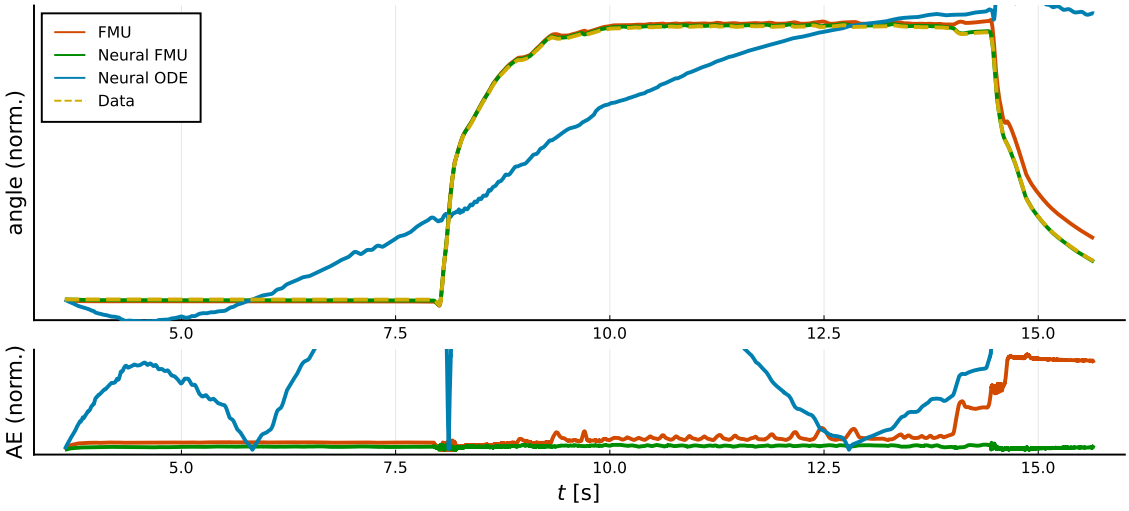


Figure 8.26: The ECM rotor angle on the testing scenario, comparing the physical (FMU), hybrid (neural FMU), and ML (neural ODE) model. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

The performance of the hybrid model on testing data is similar to training data, which is a positive indicator for proper generalization. As already mentioned, the trajectories are less similar (nominal) than they appear at first glance, and the inputs also differ between training and testing. In Fig. 8.27, the rotor speed on training data can be examined.

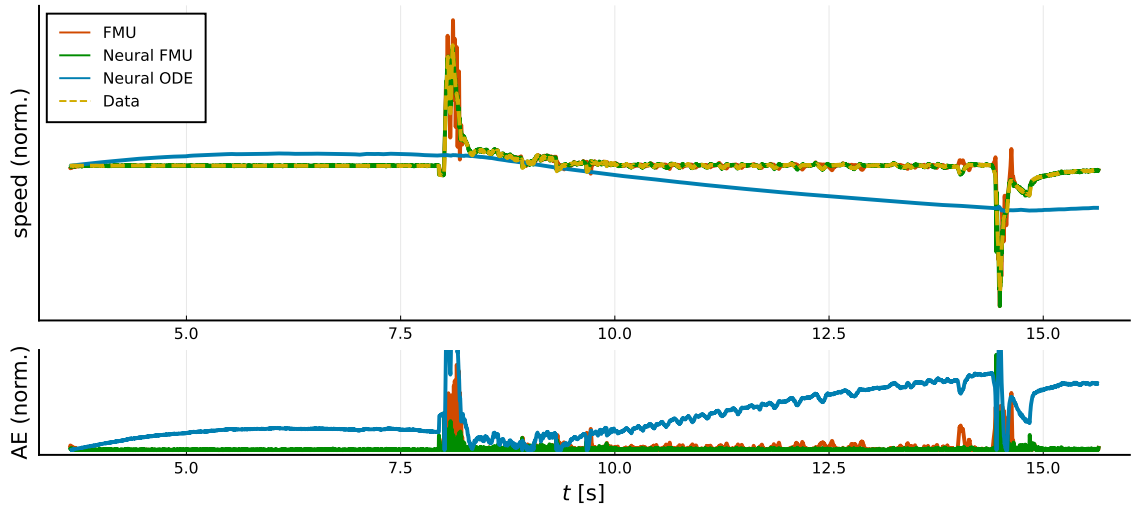


Figure 8.27: The ECM rotor speed on the testing scenario, comparing the physical (FMU) and hybrid (neural FMU) model. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

By evaluation of the quantitative results in Tab. 8.5, we find that the accuracy is even higher on testing than on training data for the rotor angle, however, not for the rotor speed. Because the original simulation model (FMU) performs worse on average, the improvement factors for the neural FMU are greater, and we find a gain in accuracy of factor ≈ 5.5 for the angle and ≈ 2.5 for the speed. This is because the investigated data segment is shorter for testing compared to training, and includes proportionally more parts with high dynamics. On segments with low dynamics, the FMU already performs well.

Table 8.5: Comparison of the solver steps (*stiffness*) and the MAE for the different models on testing data. Errors are normalized for reasons of IP.

Model	MAE (angle)	MAE (speed)	Acc. steps	Rej. steps
FMU (physical)	0.0133	0.0099	263,870	29,607
Neural FMU (hybrid)	0.0024	0.004	127,873	3,217
Neural ODE (ML)	0.1543	0.1447	200	14

Summary
With this example, we have validated that hybrid modeling is also possible and valuable in demanding industrial environments (e.g., with a largely unknown simulation model in form of a FMU). Although the actual equations are not visible, a hybrid model based on the existing simulation model is able to achieve more accurate results by several factors. In addition to the quantitative improvements, we found valuable qualitative improvements (e.g. turning in the right direction), and improved numerical properties of the resulting right-hand side. At the same time, the application of hybrid modeling is sufficient for challenging industrial conditions,

such as extreme data poverty due to costly demonstrator development and operation. Through the targeted incorporation of abstract prior knowledge (like SOP), training based on this minimalist data foundation is nevertheless successful.

We verified accurate simulation results on testing data. An interesting future direction for this use case is the extension of the training and testing datasets, for example, to include different braking scenarios like strong braking involving the anti-lock braking system (ABS), or braking during driving around a corner.

Contribution 16 *Using two different examples, the ECM and the SCARA, we showed that hybrid modeling can be applied to significantly improve the accuracy of simulation models in engineering applications in the field. For this purpose, we only required a tiny amount of data, for both use cases only a single measurement trajectory each. This motivates the usage of hybrid modeling for industrial applications, where data is costly to generate (e.g. building prototypes, taking measurement trips).*

8.4 Medical engineering: Pig cardiovascular system (PCS)

As briefly highlighted, the third experiment for validation is very different, compared to the ECM and the SCARA. Here, the goal is to increase *computational performance* instead of *accuracy*. If a simulation model with sufficient accuracy is available, it can be used for generation of data to train a more light-weight model (the *surrogate*) for similar — in terms of prediction accuracy — system behavior. Due to the smaller dimension of the surrogate (fewer states, fewer equation evaluations, etc.), an exact fit is generally not possible. However, this is in general not a problem in such applications, as the model for data generation is accurate beyond the requirements.

At this point, one can rightly question why the amount of data is limited if a simulation model is available that can be used to generate any amount of data. In the following, we introduce the specific use case and show why the amount of data can also be severely limited in *such* applications.

8.4.1 Introduction

In addition to the rather weak argument that the generation of data takes time, there is a much more obvious limitation in the presented application: the execution of the high-resolution model already *requires* data. The simulation model consists of submodels for arteries, organs and vascular beds, but an essential submodel is deliberately not modeled: The heart. The characteristic and patient-specific flow curve (the *cardiac output*) is based on a measurement of the real system and is input to the model. The direct consequence of this is that only as much training data can be generated as there is real measurement data available. The presented use case was performed in cooperation with *Philips Research* in Eindhoven in parallel to the *UPSIM* project.

Use case

As already introduced briefly, the goal within this use case differs in the sense, that a simulation model with sufficient accuracy is already developed and no further gain in accuracy is required. However, the *simulation performance* is insufficient, for example the simulation model can not be deployed on specific hardware (like a budget desktop computer or a smartwatch) because of the restricting requirements w.r.t. computational cost and model size. This is a typical scenario for models from computational fluid dynamics (CFD), that tend to cause high computational requirements because of computing QoIs at many spatial distributed discretization points. The model investigated falls in this class, the use case is captured in Fig. 8.28.

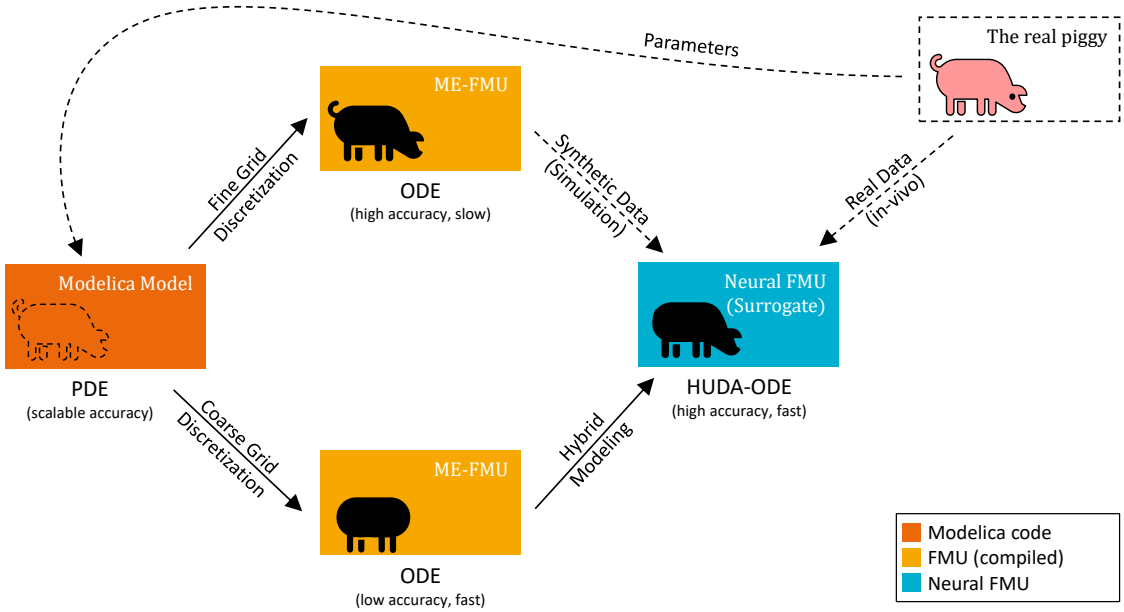


Figure 8.28: The use case of the PCS. A PDE-model is created and parameterized on the *patient* (the real piggy). Two structurally similar ODE models based on the same PDE model are generated by applying a fine-grid and a coarse-grid spatial discretization. The fine grid model (accurate, but slow) is used for data generation, whereas the coarse grid model (fast, but inaccurate) is used as core for the hybrid model. The hybrid model is trained on a mix of real data (input) and synthetic data generated by the fine grid model.

Starting with the PDE model of the PCS, two different models (in form of FMUs) can be generated. The first model is referred to as *fine grid model* and features a reasonable discretization, as proposed and validated in the corresponding publication [153]. The second model, referred to as *coarse grid model*, is generated by reducing the number of spatial discretization points to the very minimum — meaning only a single discretization point to compute the arterial cross-section and mass flow rate for any of the 23 artery segments each. Whereas this model still incorporates the physical equations of the fluid flow, the simulation results no longer reflect the real system in a recognizable way. On the other hand, it provides superior simulation performance compared to the fine grid model, as further compared at the end of the experiment (s. Tab. 8.9).

Physical model
The physical model of the PCS describes the arterial side of the cardiovascular system of a pig. To allow for object-oriented modeling of the system with Modelica, we mix the modeling paradigms of Charlton *et al.* [154] and Audebert *et al.* [153] to obtain a proper simulation model. A detailed, mathematical investigation of the model is omitted here for the reason of not being focus within this work. The model consists of the following major parts:

Arteries The arterial tree is modeled consisting of 23 arteries and artery segments, that feature a special 1D kinetic scheme in form of a PDE, including the change in diameter of the arterial cross-section caused by pressure. For spatial discretization, a special kind of finite differences up-stream scheme [153] is applied. The arterial tree is visualized in Fig. 8.29.

Branches Arterial branches are modeled without losses and preserving w.r.t. physical conservation variables (sum of inflows equals sum of outflows, pressure is the same, density is constant).

Heart Instead of modeling the heart as autonomous physical system as in Audebert *et al.* [153], it is simplified as inflow boundary condition, enforcing a mass flow at the arterial root. This is because in this application, mass flow at the arterial root is known from in-vivo sensor measurements and should therefore be incorporated as simulation input.

Vascular beds The 10 vascular beds are modeled as three-element Windkessels[6]. Instead of connecting the vascular beds to the *vena cava*, which is connected to the right atrium of the heart as in Audebert *et al.* [153], diastolic blood pressure is assumed after the Windkessels. This modeling pattern allows for only modeling the arterial side of the cardiovascular system, without significant limitations in terms of the informative value of the model [154].

Liver The liver is simulated by six interconnected Windkessels, pairwise describing the three lobes of the pig liver. The masses (and therefore the volumes) of the liver lobes are controllable via inputs, therefore a surgical reduction of the liver (*hepatectomy*) can be simulated with this model [153]. However, the liver mass is kept constant during this experiment and was only used to validate the correct modeling during physical model development.

As mentioned, for the PCS and PDE-based models in general, the spatial discretization can easily be changed. Changing the number of discretization points directly influences the number of states in the final system. This way, creation of a *coarse-grid* model based on a *fine-grid* model is straightforward. However, the discretization influences — besides the result quality of course — the stability and stiffness of the system, which is also briefly investigated as part of the results.

[6]Windkessels are typical boundary conditions in the medical domain, the three-element version equals an RCR equivalent circuit model.

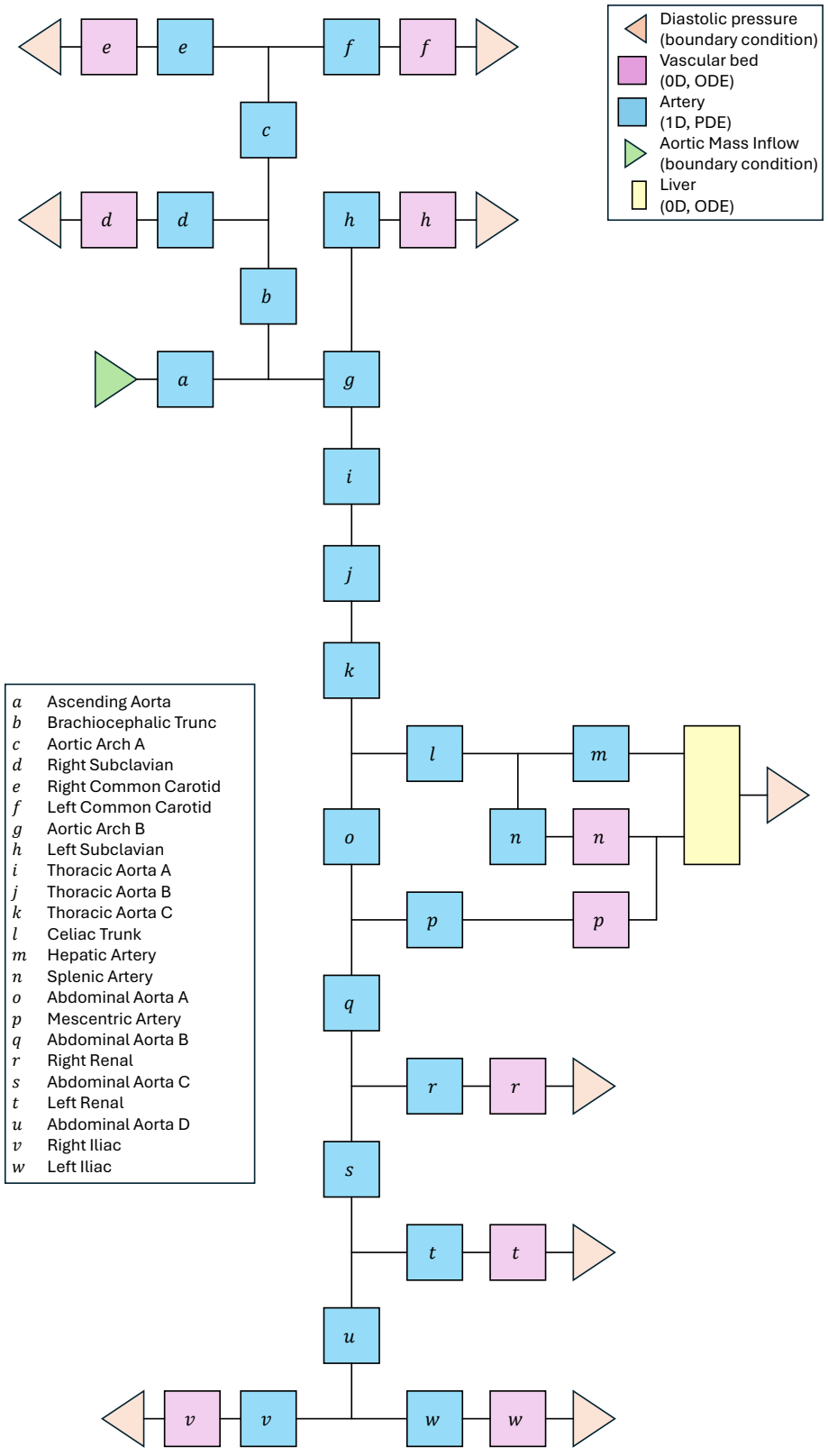


Figure 8.29: The arterial tree of the pig model. The structure is based on Audebert *et al.* [153], but modified to reflect the introduced modifications. The model consists of 1D arteries (blue), 0D vascular beds (violet), and the liver (yellow). The mass inflow boundary condition is given by the heart (green) and pressure boundary conditions (orange) are deployed to model the vascular beds.

Data

The data used was not generated in connection with the project or this work, but was kindly made available from an existing dataset. Data was recorded in-vivo from a Yorkshire pig of $\approx 50\,\mathrm{kg}$ mass through the operative insertion of various sensors. Sensors for measuring the blood flow were placed at the Aorta, the Hepatic Artery, the Portal Vein, the Common Carotid Artery and the Femoral Artery. For the considered experiment, three measurements from different times of the day (featuring slightly different heart rates) are used for training, testing and validation. Unfortunately, key variables such as heart rate and blood pressure are very similar in all three measurements, for a critical validation larger differences in the datasets would be beneficial to investigate the generalizability of the hybrid model. It is therefore decided to make well-founded modifications to the real measurement data in order to depict greater heterogeneity (*data augmentation*). The considerations and measures are described below.

In Armstrong *et al.* [155] pigs are examined under different levels of exercise. Various cardiovascular parameters are measured while the pigs walk or run on a treadmill at variable speed. It is further observed, that the stroke volume per heart beat is constant, so the cardiac output is primarily driven by the increasing heart rate at higher treadmill speeds, and that the blood pressure stays approximately the same (only a slight increase during extreme activity). Some cardiovascular parameters from this work are summarized in Tab. 8.6.

Table 8.6: Cardiovascular parameters (mean and SD) for pigs running on a treadmill. From Armstrong *et al.* [155], values marked with "*" are estimated by evaluation of a result figure.

Treadmill speed [km/h]	Mean aortic pressure [mm Hg]	Cardiac output [l/min]	Heart rate [1/min]
0.0	115 ± 8	2.1 ± 0.5	66 ± 8
4.8	115 ± 8	$4.75^* \pm 0.45^*$	$177^* \pm 19^*$
8.0	115 ± 8	$6.76^* \pm 0.75^*$	$207^* \pm 29^*$

Based on these medical observations [155], we can augment the existing real dataset by adjusting the heart rate to generate training data (easy exercise), validation data (no exercise) and test data (moderate exercise), compare Tab. 8.7. At this point it is important to emphasize that only the input signal (the aortic output) is changed and that this change leads to highly different, characteristic flow and pressure curves in the remaining arteries, as we will investigate as part of the experiment. Furthermore, we test against a dataset that explicitly checks the extrapolation, i.e. the behavior under a heart rate above the one known from the training dataset. This is a suitable measure to critically validate the extrapolation capability of the models.

Table 8.7: Modifications to the heart rate to further differentiate the three data records.

Dataset	Heart rate modification
Original	0 %
Training	+121.25 %
Validation	−17.5 %
Testing	+158.75 %

8.4.2 Preliminary examinations

Before introducing the actual experiment, some preliminary considerations are made.

State dimension
Compared to the ECM, the PCS features a significantly higher state count (4 vs. 60). Systems with large state count are more expensive[7] to solve, and SA is in particular more expensive. Even if reverse methods like the continuous adjoint SA provide sublinear cost w.r.t. parameters, Jacobians like $\boldsymbol{A}$ (size corresponds to the number of states) for the right-hand side (here: a FMU) need to be sampled. Here, this results in cost for SA that is super-linear w.r.t. the number of FMU states and sublinear w.r.t. the number of ANN parameters.

Solver / Stiffness
The model can be considered *stiff*. Following the stiffness definition in Sec. 6.3.2.2, the model is suboptimally solvable with an explicit solver, like e.g. the explicit Runge-Kutta method *Tsit5* [133]. Therefore, *FBDF*, an implicit multi-step BDF solver, is applied — featuring a significant better performance under the same tolerances, compare Tab. 8.8.

Table 8.8: Comparison of the inference times, solver steps (*stiffness*), and Jacobian evaluations of the fine-grid model for different solvers. A single heart beat of around 0.75 s is simulated. The absolute and relative tolerances are 10^{-7}, larger tolerances terminate the simulation (the sensitive algebraic loops cannot be solved).

Solver	Times [s]	Acc. steps	Rej. steps	Jacobians
Tsit5	3.95	11,400	984	0
FBDF	1.76	304	24	11

Sensitive algebraic loops
The system contains algebraic loops, that are solvable during regular simulation. However, within the hybrid model, states may be manipulated in a way that the loops cannot be solved anymore. This can be because either the loop has analytically no solution anymore, or the initial values are insufficient to iteratively find a solution

[7] Of course, if the numerical properties (e.g. stiffness) are the same for both systems.

(e.g. via Newtons method). While the former requires additional measures[8], the latter problem can be mitigated by making only small changes to the system. Small changes can be accomplished by doing small optimization steps and clipping the gradient (slowly changing the parameterization), or small finite differences sampling steps (changing the state and parameters only a little).

8.4.3 Experiment

The goal of the following experiment is to increase the computational performance of the hybrid simulation model, compared to the fine-grid model, while maintaining *similar* accuracy. The quantitative requirement is to keep an average deviation of $< 10\,\%$ for all trajectories — so along the entire cardiovascular system — on training data. On the plus side, the surrogate simulation time should be reduced to $< 1\,\%$ of the simulation time of the original simulation model.

Further, because the goal of a surrogate is to replicate the original model behavior, we only validate against data from the fine-grid model and not data from the real system, that is not available in detail for all arterial segments anyway. Comparison against in-vivo data is part of the validation of the *physical model*, which is not part of this work. In this experiment, we compare the hybrid surrogate performance to the coarse- and fine-grid, as well as a ML model.

Hybrid model architecture

Despite the huge differences in between the use cases, the hybrid model of the PCS (s. Fig. 8.30) is very similar to the ones that are used for the ECM and SCARA, which shows the broad applicability of the approach. Comparing to the ECM hybrid model, the major difference is that no delay correction for input signals is used. As for the SCARA experiment, the output function $\boldsymbol{g}$, as well as the event functions $\boldsymbol{c}$ and $\boldsymbol{a}$, are inherited from the FMU (compare Fig. 8.8).

The loss contains pressures and volume flows of all arterial segments, so all states corresponding to the 1D submodel. To promote equal fitting of all artery segments, the nominal volume flow and pressure heavily varies between the segments, min/max normalization is applied.

Hyperparameter optimization

As for the previous experiment, the Hyperband resource within the individual runs is rescaled, so that the most expensive run within the entire optimization has a budget of 200 s (instead of 81 s) of training data, independent of the chosen batch element duration. We train the proposed hybrid model under the hyperparameters as introduced in Sec. 8.1.3, however, extend the hyperparameter space by the following parameters:

Architecture Two different architectures are investigated: An architecture that is manipulating only the *arterial* states by the ANN (as introduced in Fig. 8.30) and alternatively an architecture that is manipulating *all* states (so arteries as well as the terminal Windkessels).

[8] One could think about explicit regularization in order to keep a known loop solvable, see Sec. 9.3.

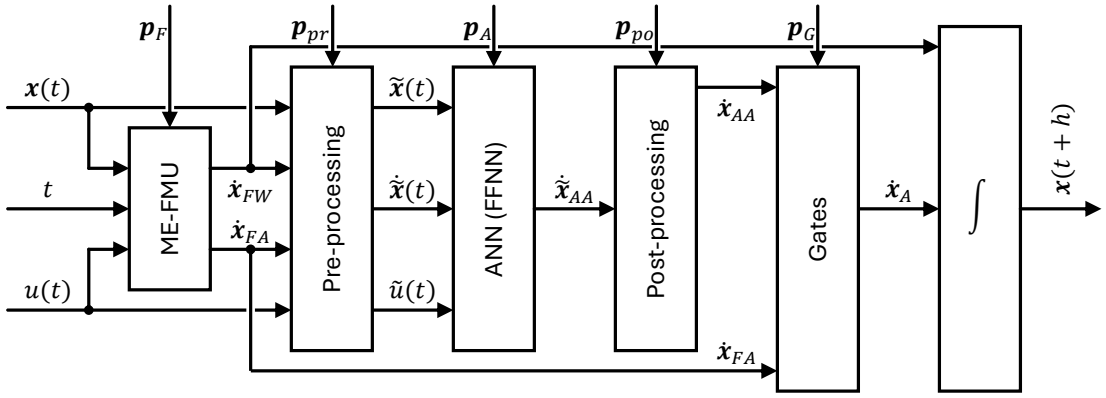


Figure 8.30: The function $\boldsymbol{f}$ (and the ODE solver) of the HUDA-ODE for the hybrid PCS model. The input $u(t)$, time t and state $\boldsymbol{x}$ are passed on to the FMU, that computes a state derivative, that is separated into the arterial state derivatives $\dot{\boldsymbol{x}}_{FA}$ and the Windkessel derivatives $\dot{\boldsymbol{x}}_{FW}$. The original state $\boldsymbol{x}$, the input $u(t)$ and the state derivatives from the FMU are pre-processed and the FFNN calculates the derivative for the arterial states, that are post-processed to $\dot{\boldsymbol{x}}_{AA}$. The *gates* block determines the proportionate composition of the hybrid model arterial dynamics $\dot{\boldsymbol{x}}_A$ from the ANN dynamics $\dot{\boldsymbol{x}}_{AA}$ and the FMU dynamics $\dot{\boldsymbol{x}}_{FA}$. The remaining state derivatives of the Windkessels $\dot{\boldsymbol{x}}_{FW}$ are forwarded together with $\dot{\boldsymbol{x}}_A$ to the ODE solver and numerically integrated to obtain $\boldsymbol{x}(t+h)$. The physical parameters for the FMU $\boldsymbol{p}_F$ are assumed to be known, whereas the parameters for the ANN $\boldsymbol{p}_A$, the gates $\boldsymbol{p}_G$, and pre- $\boldsymbol{p}_{pr}$ and post-processing $\boldsymbol{p}_{po}$, are determined by optimization (training).

Loss portion Because the loss function consists of 36 individual values, which means that a lot of information would have to be compressed in the gradient, only a subset of the quantities of interest is rated as part of the loss function. The subset of quantities is chosen randomly, the size of the subset is varied between 10 % and 100 %.

Gradient clipping The PCS is very sensitive w.r.t. changes in the dynamics. Rapid changes lead to non-solvable algebraic loops. Therefore, large parameter gradients are clipped by defining an upper-bound that is varied between 10, 100 and 1,000.

Baselines

Similar to the previous validation experiment, we baseline against a pure ML model as well as against the pure physical models. The difference here is that there are two physical models: The fine- and coarse-grid model. Depending on the discipline, one of the two models is compared against the hybrid model. While it is clear that the fine-grid model cannot be surpassed in terms of accuracy (as it serves as GT with a deviation of 0), the simulation time should be significantly undercut. Comparing to the coarse-grid model, on the other hand, the hybrid model should be significantly more accurate and a small increase in computing time is acceptable. On the ML side, we again use a neural ODE and expand the hyperparameter space accordingly as in the ECM experiment. Similarly to the ECM example, we do not expect meaningful

results, because the amount of data, training resources, and the data exploration of the state space are very limited.

8.4.4 Results & discussion

In the following, we discuss the results of the best hyperparameter run.

Training
We start by checking the compliance of the requirements on the training data. First, we investigate the simulation trajectories (s. Fig. 8.31). We find quantitatively good results for the hybrid model, all simulation trajectories for the hybrid model are within the average accuracy requirement of 10 % (compare the subplot titles). In fact, we find an average deviation of only around 5 % for the arterial cross-section and flow, which significantly undercuts the accuracy requirement. This is particularly impressive in view of the fact that the coarse-grid model (red), which serves as the foundation for the hybrid model (green), exhibits very large deviations compared to the fine-grid model (yellow). The neural ODE (blue) again shows very large deviations, which we therefore do not examine further in terms of quality. The good performance on the individual artery segments of course results in a good overall performance of the hybrid model, which can be quantitatively verified by investigation of Tab. 8.9.

Table 8.9: Comparison of the inference times, solver steps (*stiffness*), and average errors for the different models on training data. Errors are normalized for reasons of IP.

Model	Sim. time [s]	MAE (vol. flow)	MAE (area)	Acc. steps	Rej. steps
FMU (fine-grid, phy.)	269.27	0.0	0.0	5,859	517
FMU (coarse-grid, phy.)	0.045	1.0502	0.2977	222	18
Neural FMU (hybrid)	2.215	0.0517	0.0497	507	184
Neural ODE (ML)	0.218	2202.2969	12467.7671	44	0

Another, equally important part of the validation is checking the requirement regarding the computing time. The first important observation is that the fine-grid model has significantly longer computation times than the remaining models. While the coarse-grid model is the fastest, the neural ODE and the hybrid model require more computing time. This can be attributed to a more complex right-hand side and unused optimization potential in the software implementation. However, the overall simulation performance of the hybrid model features a speed-up of factor ≈ 122, so that we find a hybrid model that fulfills all requirements by far.

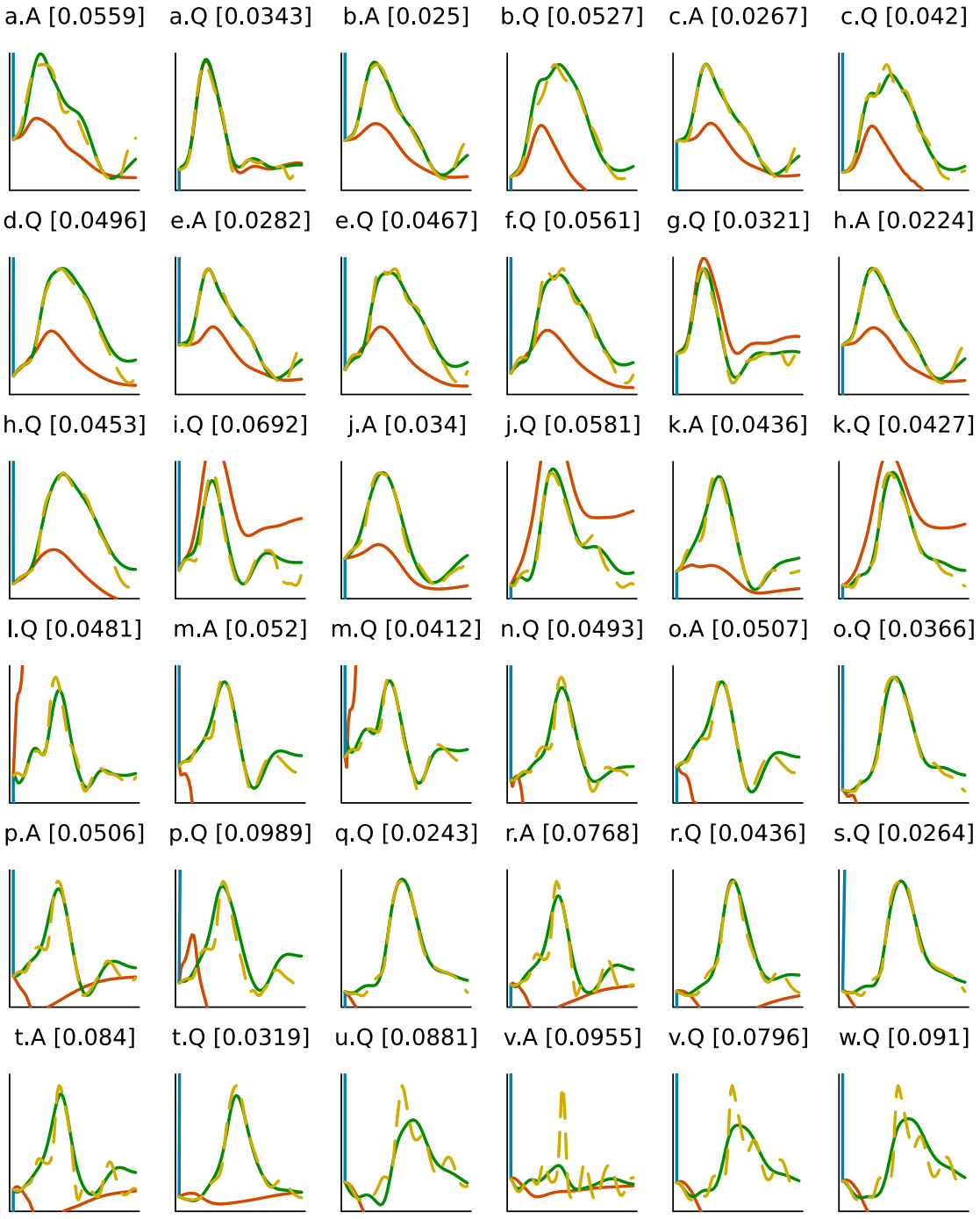


Figure 8.31: Training results of the PCS experiment. The plots show the arterial cross-section area (A) and volume flow rate (Q) for the 23 arterial sections (a-w) on the vertical axis over the time of a single heart beat on the horizontal axis. The fine-grid model (yellow, dashed) is compared to the coarse-grid model (red), the ML model (blue, clipped), and the hybrid model (green). The subplot titles identify the arterial section and quantity, and give the MAE for the corresponding quantity. All quantities are normalized for IP reasons.

The results on validation data are unremarkable (see Sec. A.11), so the results on test data are examined below.

Testing

Before investigating testing results, it is important to state that the main goal of a surrogate is not to extrapolate well. In general, surrogates are trained on a sufficiently large dataset, and the goal is to deliver good results *in-distribution*, meaning for samples that lay in between the training data. For testing, however, we investigate the model performance on *out-of-distribution* data, so samples, that are beyond training data. In a nutshell: The hybrid surrogate is trained on data of a pig doing easy exercise (*walking*), and is tested on data of a pig doing moderate exercise (*running*). This is a very critical investigation, as new, more dynamic regions are simulated. We find once again that such a task cannot be accomplished with a minimal amount of data, as is the case here, for a pure ML model. We start by investigating the trajectories on testing data, that are very different compared to training data, see Fig. 8.32.

By completing the results of the plots with Tab. 8.10, we find a very good testing error of around 11 % for the arterial cross-section, as well as the flow. This can be considered an excellent result, keeping in mind that we sampled far out-of-distribution for testing. This clearly shows that the equations of the coarse-grid model — the *heart* of the hybrid model — are being reused very efficiently, since no significant extrapolation could be expected from learning on the tiny amount of data.

Table 8.10: Comparison of the inference times, solver steps (*stiffness*), and average errors for the different models on testing data. Errors are normalized for reasons of IP.

Model	Sim. time [s]	MAE (vol. flow)	MAE (area)	Acc. steps	Rej. steps
FMU (fine-grid, phy.)	298.726	0.0	0.0	5,896	457
FMU (coarse-grid, phy.)	0.044	1.1891	0.3336	179	10
Neural FMU (hybrid)	0.621	0.1076	0.112	106	18
Neural ODE (ML)	0.044	1883.1599	10660.812	39	0

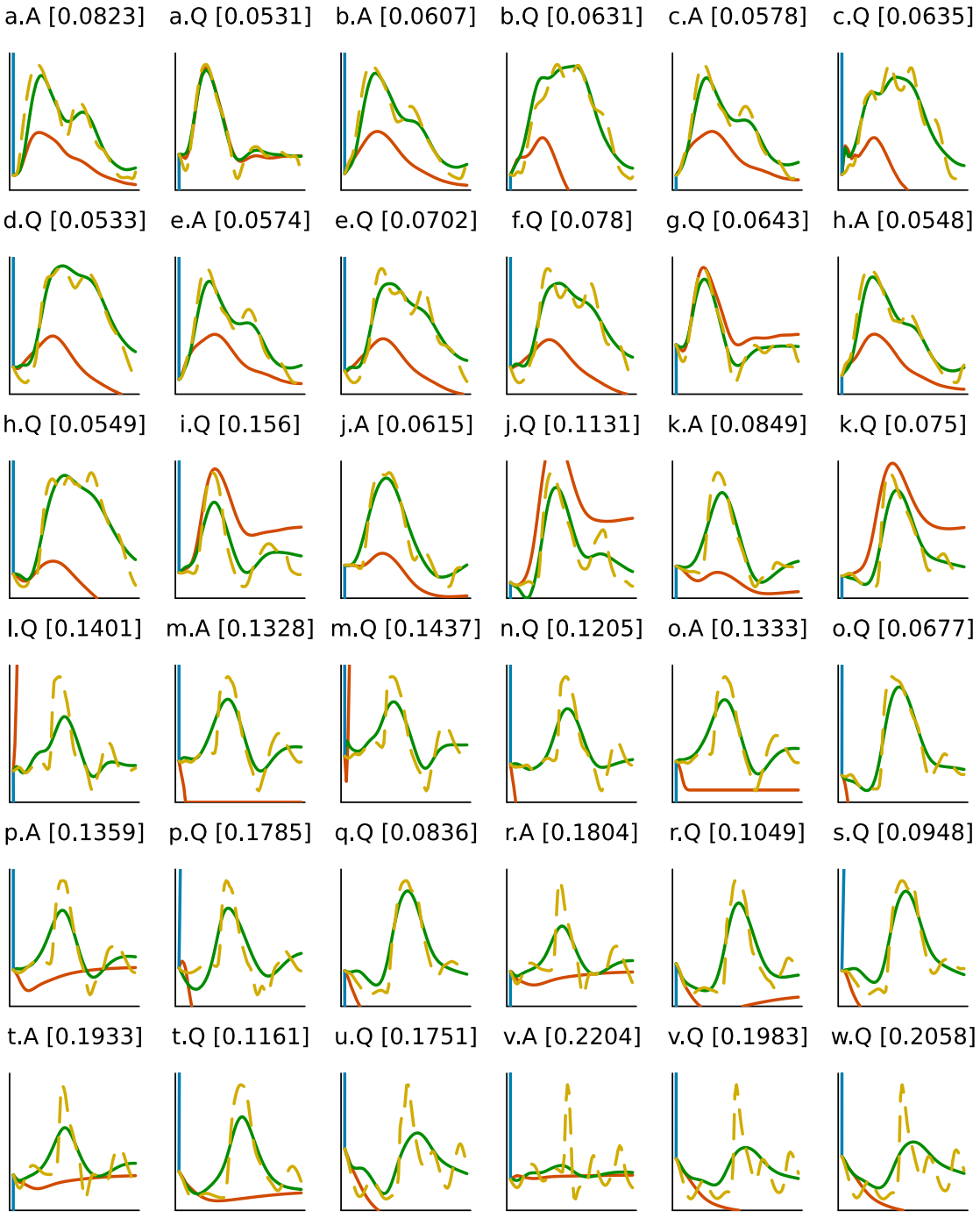


Figure 8.32: Test results of the PCS experiment. The plots show the arterial cross-section area (A) and volume flow rate (Q) for the 23 arterial sections (a-w) on the vertical axis over the time of a single heart beat on the horizontal axis. The fine-grid model (yellow, dashed) is compared to the coarse-grid model (red), the ML model (blue, clipped), and the hybrid model (green). The subplot titles identify the arterial section and quantity, and give the MAE for the corresponding quantity. All quantities are normalized for IP reasons.

Summary
To summarize the quantitative results, the hybrid model is able to provide much better average accuracy compared to the coarse-grid model (factor ≈ 20 on flow, ≈ 6 on area) and the pure neural ODE. In summary, the average deviation compared to the fine-grid model is around 5 % on area and flow on training data, and for out-of-distribution testing only around 11 % on average. At the same time, we find a valuable reduction of the computation time compared to the fine-grid model (factor ≈ 122 on training, factor ≈ 481 on testing).

Contribution 17 *Using the example of a surrogate application from a real medical technology environment, we have shown that we can achieve a massive speed-up by applying hybrid modeling. At the same time we are able to satisfy the given accuracy requirement. What makes this especially impressive is the fact that the hybrid model is trained only on a single heart beat of measurements. Finally, performance of the hybrid surrogate in out-of-distribution scenarios is significantly better compared to pure ML, because the broadly valid physical equations inside the hybrid model extrapolate well. This motivates for the use of such hybrid surrogates in applications with only very limited access to patient-specific data — which is a great relief in the clinical field.*

8.5 Further examples

Besides the ones introduced, the presented methods where applied to many further use cases successfully. They are not presented here in full length, but strengthen the relevance of the introduced methods:

Human cardiovascular system In Thummerer *et al.* [92], a hybrid model for the human cardiovascular system is designed based on a simple, but fast ODE simulation, that explicitly neglects the elasticity of the arterial walls. Based on data from a high-fidelity CFD simulation, a hybrid model is developed, featuring sufficient accuracy with a speed-up of factor $> 3,000$ compared to the full-scale CFD model.

Vehicle simulation In Thummerer *et al.* [83], a vehicle longitudinal dynamics model, featuring many challenges like discontinuity, controllers, characteristic maps and hysteresis, is improved based on real data. The final model, that was further enhanced for a workshop [97] at the *JuliaCon 2023* at the Massachusetts Institute of Technology, is able to improve prediction accuracy by a factor of 22 (max deviation factor 11). A preliminary version of this model was also presented at a workshop [156] at Bosch in Renningen.

Hydraulic excavator arm In Thummerer *et al.* [93], a hybrid model for a part of an hydraulic excavator arm is developed. The model is able to learn for an improved friction model based on a simplified (viscous) friction model, as well as the completely undamped system. Especially interesting is, that the experiment itself was performed at ESI (ESI Group), using the methods and software developed within this thesis.

Stick-slip pendulum In the first publication on the topic of integrating FMUs and ANNs [129], a simple translational pendulum in form of a FMU is augmented by an ANN, again to learn for stick-slip friction. The hybrid model was able to improve prediction accuracy, however four years later these results are outdated and applying the methods derived within this thesis would allow for any exact approximation.

Chapter 9

Conclusion

This work concludes in the traditional way with a summary and an outlook for future work.

9.1 Summary

Recalling the previous chapters, we realize that after we introduced the topic of hybrid modeling and SciML in general, we have gone through the entire development process of a hybrid model once.

We started by introducing some mathematical **basics**, that are necessary for hybrid modeling. We investigated different kinds of equation systems, and focused especially on the ODE. Based on that, we thought about how SA works for the solution process of ODEs, which we identified as the prerequisite for training of hybrid models. Because hybrid modeling involves physical modeling and ML, we defined some common terminology.

Equipped with mathematical knowledge, we started our journey to **create** a hybrid model. We investigated the state of the art and came to the conclusion, that we need to decide on *which types* of models we want to combine, before discussing *how* we can combine them. We defined the HUDA-ODE and concluded, that they fit a wide variety of typical engineering models. Based on the HUDA-ODE, we investigated all possible combination schemes and developed a strategy to *learn* for connections between models.

Now that we have a hybrid model, we want to evaluate it. However, **initialization** of parameters and states is crucial to gain a solvable model. Especially at domain boundaries, where physics and ML come together, signals must be carefully transformed by pre- and post-processing operations. By introducing *gates*, we can further weaken the impact of suboptimal initialization. These steps allow for simplification of the initialization challenge and reuse of existing methods.

Because a well-initialized hybrid model is able to simulate, but not to make better predictions compared to a conventional model, we need to **train** it. Training of hybrid models is challenging, and different strategies depending on requirements are proposed and compared.

When training alone is not enough to achieve the goal, **regularization** techniques can be applied to improve training performance, as well as to find better local optima. We derived special regularization methods, based on problems we

faced during working with real simulation models in engineering. This covers eigenvalues, errors and event indicators.

At this point, the problem of developing a hybrid model (within this thesis) can be seen as solved on a mathematical level. However, some **real world** aspects are not captured by mathematics, e.g. software and the model format. Simulation models can not be transferred as written systems of equations in general. We therefore built a bridge involving the model exchange format FMI between mathematics and industrial application, that allows for the reuse of industrial models (FMUs) within the derived pool of methods.

The entire process of hybrid model creation needs to be **validated**. During writing this thesis, multiple use cases where worked on. Three very special ones — the selective compliance assembly robot arm (SCARA), the electronically commutated motor (ECM), and the pig cardiovascular system (PCS) — are investigated in detail as part of a critical validation of the methods and techniques developed within this thesis. We were able to outperform physical modeling many times over regarding accuracy (ECM and SCARA) and computational performance (PCS). An important insight is that efficient hybrid modeling still requires some clever decisions by an engineer to achieve very good results on use cases with very little data. We further showed, that pure ML is not a valuable option for real world engineering with very limited access to data.

Finally, we want to close this thesis by concerning whether the actual *goal of this thesis* is reached. The initially formulated goal was:

To advance the applicability of hybrid simulation models based on SciML in the industrial field, by addressing and solving the challenges that typically emerge during working on different engineering use cases.

To answer this, we can revisit the challenges we highlighted during deduction of the hybrid modeling paradigm:

Solvability We found, that state-of-the-art hybrid modeling is not able to make statements regarding the solvability of the derived system of equations. We discussed possibilities to combine models, and investigated the solvability of the resulting models. Based on that, we proposed the *PSD* model architecture, that leads to a solvable combined model and can even be used to train for various model architectures. Further, we also provide an initialization strategy for the connections within the hybrid model, which, for example, allows for the reuse of the initialization routine of the physical submodel.

Discontinuity Similarly to the *solvability*, handling of discontinuities for combined models is not discussed in the general context. We therefore derived a strategy to handle combination of discontinuous models and derived a combined event handling strategy, that may even be simplified for special edge cases.

Interfaces We found, that many types of models are used in hybrid modeling, which results in different interfaces that need to be considered during model combination. To derive a broadly applicable solution for this, we derived the HUDA-ODE that is able to capture a variety of engineering and ML models.

Implementation To further increase the re-use of methods, we provide a software implementation together with the definition of the HUDA-ODE, that

allows for re-implementing various models from physics and ML. However, only the model *interface* needs to be implemented, algorithms for inference, combination and training are applicable for the entire range of HUDA-ODEs out-of-the-box.

Tooling We found, that physical modeling and ML require different tools. Therefore, we proposed a special development cycle to work with existing simulation models from common engineering tools based on FMI.

Instability More complex hybrid models tend to become instable, or inefficiently solvable in general. We proposed the eigen-informed regularization technique, that is not only capable of stabilizing hybrid models, but can further be used to control stiffness, frequencies, damping and other properties from systems and control theory. For models that suffer from event chattering (which can be interpreted as discrete counterpart of stiffness), also a regularization strategy was presented.

Assertions We further found, that model development involves the definition of model assertions to obtain a maintainable and manageable system model. These assertions may be violated by the hybrid model. Here, we propose a regularization strategy to prevent violation of model assertions during training.

We conclude, that the open challenges have been solved or significantly mitigated and that hybrid modeling in industrial applications is significantly simplified or — depending on the use-case — even made possible using the methods introduced. Of course, many specific questions still remain in special applications, but a targeted attempt has been made to make a significant contribution to solving the most common problems. This was demonstrated by successfully processed use cases, that where not solvable before without applying the presented methods. Finally, the fulfillment of the goal of this thesis can be justified by the interest of industry[1] in neural FMUs.

Hybrid modeling — and SciML in general — is a trade-off between incorporation of knowledge and the availability of data. In a world with unlimited resources, we can do ML and can refrain from building in how we think we think *(recall* The Bitter Lesson *[1]). However, if the amount of data or the available computational power is not enough — or if we simply want to protect the environment a little — we can incorporate a priori knowledge in our ML model and make it fast, efficient, and actually* scientific*. We do not have to learn* everything *from data, we have already gathered a little knowledge over time. Let's use it!*

9.2 Limitations

Before finally investigating future work, we want to critically summarize the core limitations of the presented work.

[1]Several industry partners from different engineering domains tested or are currently testing the use of neural FMUs in their own applications and workflows.

Validation of fully learnable architecture at real world application
In the validation experiments, we kept parts of the architectures trainable by introducing gates. This is a meaningful measure and reflects the mindset that it is beneficial to incorporate prior knowledge structurally into the model, without creating unfounded and unnecessary degrees of freedom (parameters). However, it will be interesting to examine, how *fully* learnable architectures (*PSD*) will perform in challenging scenarios with hundreds or thousands of possible connections. The validation of fully trainable architectures in applications of industrial scale was not validated within this work.

Validation of eigen-informed regularization at real world application
Whereas this work introduced the concept of eigen-informed regularization and showed the applicability on the bouncing ball example, a critical validation of this approach for a real world problem has not yet been performed. This is because for the validation use cases within this work, eigen-informed regularization is not a meaningful measure, because no significant a priori knowledge w.r.t. to eigenvalues was available. Unfortunately, no other real world industrial use case was available at the time of writing, but a detailed examination under industrial boundary conditions is one of the key points of future work.

Further, eigen-informed training requires *nested AD*, meaning partial derivatives w.r.t. other partial derivatives are required. This is because linearization to obtain the matrix $\boldsymbol{A}$ requires derivation of the state derivative w.r.t. the state, and additionally, operations including this matrix are derived w.r.t. parameters. Whereas this is almost straightforward from a mathematical perspective, the implementation within a software framework is not. Only a few libraries for AD, for example *ForwardDiff.jl*, support nested AD. However, *ForwardDiff.jl* features the typical limitation of forward AD to scale insufficiently with the number of parameters. However, new approaches like *Enzyme.jl*, that performs forward and reverse AD on the code level of the LLVM intermediate representation [51], [52], provide promising future perspectives here.

Implementation progress of optional FMI features
As discussed, FMI defines some of its features as optional. This naturally leads to the fact, that many exporting tools only implement the *required* subsets of the functions, and (at least for a start) skip the implementation of optional features. However, especially these optional functions, e.g. get/set the FMU memory state and getting directional derivatives, significantly improve the computing speed in practice and therefore play a key role in the applicability of the proposed methods in the real world.

9.3 Future work & open research

This thesis closes by shedding light on open questions in research, that are highly related to the investigate topics.

Dynamic state selection
Some models feature *dynamic state selection*, that allows to switch between multiple

sets of states and equations, based on the condition of the system of equations. If the system of equations becomes close to singular, a different set of states, with non-singular conditioning, is activated. This is a very common measure in multi-body physics. Switching the set of states further results in interrupting and restarting the ODE solver, as for common events, and may therefore be expensive. Further, the nominal values of states may change, which influences the absolute solver tolerance, that must be adjusted by the new nominal values.

For hybrid models, this opens up to new challenges for architectures that modify the system state or dynamics. There are three obvious workarounds to be mentioned:

1. All sets of states (and state derivatives respectively) are concatenated and used to determine the dimension of the *interface* of the physical model. Redundant entries are only listed a single time. The entire *extended state* is connected to surrounding submodels. For example, if an ANN uses the state as input, the input dimension becomes larger respectively, so that active states, as well as non-active "states"[2] are connected.

2. The discrete state (or variable), that determines the set of states that is currently active, needs also to be connected to the surrounding models. In case of an ANN, this additional input can be used to distinguish the influence of different state sets.

3. Parts of the architecture can be exchanged dynamically, depending on the current active state set. For example an ANN to manipulate the system state or dynamics can be exchanged between different active state sets.

However, the proposed workarounds need to be thought through further, implemented, tested and validated.

Tooling

The concept of HUDA-ODEs is of mathematical nature and valid beyond tool boundaries. However, we currently use FMI to transfer simulation models between modeling tool and Julia. In the future, hybrid modeling should be — wherever possible — enabled within a single tool, so a tool that is capable of providing concepts of object-oriented modeling and further features ML technologies like AD. For now, there is no such tool available at the market, that is tried and tested with broad industrial coverage. Maybe the closest we have for now is *Dyad* — formerly *JuliaSim* — (JuliaHub), which currently has a strong domain focus and is, compared other modeling tools available, not widely used yet. However, the journey of this cloud-based tool is very promising. In addition, there will still be a need for doing hybrid modeling involving black-box models (like FMUs), whenever models that are sensitive w.r.t. IP are shared between companies.

Extension to DAEs

In Sapienza *et al.* [20], a great overview of how to extend methods of SA from ODEs to DAEs and PDEs is given. Of course, this does not solve the challenge of adopting methods for other DEs completely, however it serves as valuable starting point for

[2] All "states", that are not part of the active state set, are actually no states, because they are *not* determined by integration.

the mathematical core challenge of SA. This is the basis for making considerations on how DAE can be combined in a learnable fashion.

Extension to stochastic models
In this work, *deterministic* models where focused, that predict values without uncertainty. However, for more credible simulation results, the use of *stochastic* models, that are able to cope uncertainties, and therefore randomness, seems a valuable next step.

Use of other UAs
In this work, primarily FFNNs where used as ML models. Other network structures, like RNNs are investigated from a theoretical point of view, but are not applied in the practical application. Two types of UAs are distinguished: UAs with states (like e.g. RNNs) and UAs without states (like e.g. FFNNs). The application of ML models that feature own states (e.g. RNNs or LSTM) seems not to be useful in the broad general context, because the state space of the HUDA-ODE can be augmented (adding further latent states) and paired with non-state ML models if needed, for example to add additional spatial states. The results of augmentation are similar compared to state ML models, but explainability is better and initialization more straightforward. On the other hand, investigating how other state-less UAs [157] perform as part of SciML models seems reasonable.

Explainable AI
After training a hybrid model for better predictions, the logical next step is to gain new insights based on these models. SciML captures the study of incorporation of knowledge into the model, the extraction of knowledge seems the logical next step. This is referred to as *explainable AI* [158], a promising way to increase explainability is *symbolic regression*, that approximates a model with limited explainability by a symbolic term. This is shown to be applicable to a variety of deep models [159], [160]. In addition, rule-based ML as in Heider *et al.* [161] provides a very interesting technique to discover explainable discrete dynamics. Both approaches could be paired in order to efficiently learn for explainable mixed-continuous-discrete models.

PeN-ODEs
Within the *OpenSCALING* research project, new methods for physics-enhanced neural ordinary differential equations (PeN-ODEs) are researched. From a mathematical perspective, this is an edge case of the HUDA-ODE, where exactly one physical and one ML model are combined. However, the term physics-enhanced neural (PeN)-ODEs still needs clarification and a proper definition, that will be elaborated during the project.

Loss functions on *continuous* data
Intuitively, the cost function is defined on continuous variables, so the continuous state or state-dependent variables. The natural consequence of this is that the optimizer tries to optimize the system dynamics accordingly — which is intended, of course. However, this has the consequence that discontinuities within a model are not actively used by the optimizer. At the example of the bouncing ball, the

optimizer would try to approximate the discontinuity on impact with the floor/wall with a continuous function instead of using the event function, as illustrated in Chen *et al.* [16]. While practice shows that the correct solution, including the event points, is often found nevertheless in simple cases, strategies should be developed for large simulation models with many discontinuities. An obvious but suboptimal way to deal with that issue for now is applying a *growing horizon* training.

Invertible ML models

As introduced in Sec. 3.4.4, event-handling of hybrid models including events is challenging. However, solving the nonlinear system of equations is only necessary, if the connection matrices and the submodel $\boldsymbol{s}_a$ are not invertible. If permutation matrices are used for the connections, which is common if connecting by hand or could be forced using regularization, only the non-invertible submodel $\boldsymbol{s}_a$ remains. For physical models, forcing invertibility is a severe restriction. However, if $\boldsymbol{s}_a$ is a ML model, invertible ML models, like invertible neural networks [162] could be applied. This kind of ANN is invertible by design and was developed for solving inverse problems. As a result of applying these measures, solving the non-linear system of equations during event-handling would be eliminated.

Learning for eigen-informed controllers

In Sec. 6.3, we introduced eigen-informed regularization for the regularization of the optimization of ODEs. Forcing the eigenvalues into specific positions does not only open up to stabilizing the training procedure, but also to obtain a *stable* system. In a sense, this is the core goal of the field of control theory. During development of controllers, the goal is to reach a specific target value by controlling the system, but further to reach it without destabilization of the system. In this context, it is straightforward to apply the presented hybrid modeling approach to the field of control design: Starting with a physical model of the controlled system, the ML model takes the role of a controller and is trained so that the resulting hybrid model is stable within its operation range. This is a very promising direction for future research. Also, the targeted incorporation of differentiable eigenvectors should be investigated.

Neural SE-FMUs and Neural SSPs

Since SE is a new feature that was added in FMI 3.0 and is still not widely used, the applicability of the neural FMU approach to SE-FMUs has not yet been investigated in detail. Since the SE interface has properties of both the CS and ME interfaces, neural FMUs can also be developed on the basis of SE-FMUs. However, the details of this combination still need to be researched in the future.

The standard system structure and parameterization (SSP) [163], on the other hand, has been published for some time, but is not as widespread as FMI. SSP allows the sharing of model structures and parameters in the form of several interconnected FMUs between different applications or tools. At the end of this thesis writing period, SSP version 2.0 was published very recently. As for neural FMUs, there is of course the potential to incorporate ANNs or other UAs into this model structure, too. This is also the subject of current and future research.

Dedicated neural ODE optimizers
Neural ODEs feature different attributes, compared to other machine learning problems, but are optimized using the very same optimization algorithms. Especially for hybrid models that involve complex models, gradient determination is very expensive, compared to gradient determination through "classical" machine learning models like FFNNs. This motivates for an optimizer, that gets the maximum out of the gradient information, before calculating a new one. Common methods, like e.g. Adam [98], do the exact opposite: Gradients are smoothed before applying them to the parameters. It seems a valuable direction of research to think about optimizers that respect the special needs of neural ODEs and hybrid models in SciML. Further, this directly opens to the topic of *active learning*, where specific data can explicitly be requested.

Regularization of residuals of algebraic loops
When generating code for the execution of an ODE system, algebraic loops are isolated and solved iteratively, for example by applying Newtons method. As investigated for example for the PCS, hybrid modeling may lead to algebraic loops that become harder — or not at all — solvable. The core issue here is, as for issues with model events and errors, that the optimizer is not aware of the algebraic loops. Because solving of algebraic loops corresponds to finding a zero crossing (or narrow it down), a *residual* is investigated. Finding a zero crossing is related to finding a minimum value, as we do in optimization, therefore one could think about extending a training loss function by additional terms for residuals of algebraic loops. This way, the optimizer tries to keep the system in a state that corresponds to algebraic loops that are easy to solve.

At this point, it is hard to stop writing, because the topic of SciML opens up to an incomprehensibly wide field of possibilities. Almost every algorithm from SciCo that has been developed over time can be expressed in a differentiable fashion and be used to regularize — explicitly or implicitly — the training of a (hybrid) ML model. SciML has already produced a lot of breathtaking research and will continue to do so in the future — because we are far from finished and there is still a lot to explore.

Appendix A

Appendix

A.1 Hybrid model by Schubert *et al.*

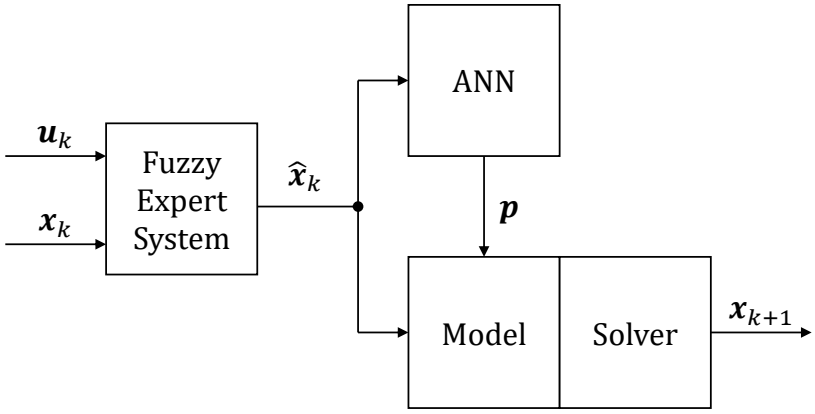


Figure A.1: Hybrid model architecture proposed by Schubert *et al.* [56] (adapted figure): Based on inputs $\boldsymbol{u}_k$ (*online data*) and states $\boldsymbol{x}_k$ (*offline data*), a corrected state $\hat{\boldsymbol{x}}_k$ is determined by a *Fuzzy Expert System*. The remaining structure resembles Fig. 3.8 [9], except the missing input for ANN and physical model.

A.2 Rediscover known models in HUDA-ODEs

To emphasize the relevance of the HUDA-ODE, it can be shown that many relevant model types can be expressed by this class. In the following, we briefly introduce model classes and give a representative model for each class.

A.2.1 Physical models: Bouncing ball 2D

In the following, the HUDA-ODE for the bouncing ball example, as introduced in Sec. 2.4, is given.

Continuous state
The continuous state is defined to be the ball positions s_x and s_y and the corre-

sponding time derivatives (velocities):

$$\boldsymbol{x}_c = \begin{bmatrix} s_x \\ \dot{s}_x \\ s_y \\ \dot{s}_y \end{bmatrix} . \tag{A.1}$$

Discrete state
The discrete state is defined to be the number of happened collisions n:

$$\boldsymbol{x}_d = \begin{bmatrix} n \end{bmatrix} . \tag{A.2}$$

Parameters
The system is parameterized by mass m, gravity constant g, radius of the ball r, energy loss by collision d, maximum number of collisions before breaking n_{max}, and the coefficients of air resistance for the two cases *ok* μ_{ok} and broken μ_{brk}:

$$\boldsymbol{p} = \begin{bmatrix} m \\ g \\ r \\ d \\ n_{max} \\ \mu_{ok} \\ \mu_{bad} \end{bmatrix} . \tag{A.3}$$

Output equation
The system output is the absolute velocity v:

$$\boldsymbol{g}(\boldsymbol{x}_c) = \begin{bmatrix} v(\boldsymbol{x}_c) \end{bmatrix} = \begin{bmatrix} \sqrt{\dot{s}_x^2 + \dot{s}_y^2} \end{bmatrix} . \tag{A.4}$$

(Continuous) state space equation
The state space equation is different for the two modes *ok* and *brk* (broken), featuring different coefficients for the air resistance:

$$\dot{\boldsymbol{x}}_c = \begin{bmatrix} \dot{s}_x \\ \ddot{s}_x \\ \dot{s}_y \\ \ddot{s}_y \end{bmatrix} = \boldsymbol{f}(\boldsymbol{x}_c, \boldsymbol{x}_d, \boldsymbol{p}) = \begin{cases} \begin{bmatrix} \dot{v}_x \\ \frac{-|\boldsymbol{v}| \cdot v_x \cdot \mu_{ok}}{m} \\ \dot{v}_y \\ \frac{-|\boldsymbol{v}| \cdot v_y \cdot \mu_{ok}}{m} - g \end{bmatrix} & \text{if } n < n_{max} \\ \begin{bmatrix} \dot{v}_x \\ \frac{-|\boldsymbol{v}| \cdot v_x \cdot \mu_{brk}}{m} \\ \dot{v}_y \\ \frac{-|\boldsymbol{v}| \cdot v_y \cdot \mu_{brk}}{m} - g \end{bmatrix} & \text{if } n \geq n_{max} \end{cases} . \tag{A.5}$$

Event condition function
An event is triggered for collisions with any of the four walls at $s_x \in \{-1\,\mathrm{m}, 1\,\mathrm{m}\}$ or $s_y \in \{-1\,\mathrm{m}, 1\,\mathrm{m}\}$:

$$\boldsymbol{c}(\boldsymbol{x}_c, \boldsymbol{p}) = \begin{bmatrix} 1 + s_x - r \\ 1 - s_x - r \\ 1 + s_y - r \\ 1 - s_y - r \end{bmatrix} . \tag{A.6}$$

Event affect function
For every collision, the collision counter n is incremented, the corresponding position is reset (to be on the collision plane) and the corresponding velocity is inverted and decreased by $1 - d$:

$$\boldsymbol{a}(\boldsymbol{x}_c^-, \boldsymbol{x}_d^-, \boldsymbol{p}, i) = \begin{cases} \begin{bmatrix} -1 + r \\ -\dot{s}_x^- \cdot (1-d) \\ s_y^- \\ \dot{s}_y^- \\ n^- + 1 \end{bmatrix} & \text{if } i = 1 \\ \begin{bmatrix} 1 - r \\ -\dot{s}_x^- \cdot (1-d) \\ s_y^- \\ \dot{s}_y^- \\ n^- + 1 \end{bmatrix} & \text{if } i = 2 \\ \begin{bmatrix} s_x^- \\ \dot{s}_x^- \\ -1 + r \\ -\dot{s}_y^- \cdot (1-d) \\ n^- + 1 \end{bmatrix} & \text{if } i = 3 \\ \begin{bmatrix} s_x^- \\ \dot{s}_x^- \\ 1 - r \\ -\dot{s}_y^- \cdot (1-d) \\ n^- + 1 \end{bmatrix} & \text{if } i = 4 \end{cases} . \tag{A.7}$$

Error function
The acceleration by air resistance must be in opposite direction to the motion of the ball (compare Sec. 6.7.3 and especially Eq. 6.38):

$$\boldsymbol{\delta}(\boldsymbol{x}_c, \dot{\boldsymbol{x}}_c, \boldsymbol{p}) = \begin{bmatrix} sgn(\dot{s}_x) \cdot \ddot{s}_x \\ sgn(\dot{s}_y) \cdot (\ddot{s}_y + g) \end{bmatrix} . \tag{A.8}$$

For readability, we allow $\boldsymbol{\delta}$ to depend on the continuous state derivative $\dot{\boldsymbol{x}}_c$. This is not a restriction, because $\dot{\boldsymbol{x}}_c$ can be substituted by the corresponding expressions from $\boldsymbol{f}$ to resolve this dependency.

A.2.2 Algebraic ML models: FFNNs

As example for a pure algebraic model, we give the HUDA-ODE of a FFNN layer in the following. A FFNN layer with m inputs and n outputs can be interpreted as fully algebraic HUDA-ODE without states and events.

Parameters
The parameters of a FFNN layer are given by the weights $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ and bias

$\boldsymbol{b} \in \mathbb{R}^n$:

$$\boldsymbol{p} = \begin{bmatrix} W_1 \\ \vdots \\ W_{n \cdot m} \\ b_1 \\ \vdots \\ b_n \end{bmatrix}. \tag{A.9}$$

Output function
The output function calculates the output $\boldsymbol{y} \in \mathbb{R}^n$ and only depends on the layer input $\boldsymbol{u} \in \mathbb{R}^m$ and parameters $\boldsymbol{p}$, and is given by

$$\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{u}, \boldsymbol{p}) = \boldsymbol{\alpha}(\boldsymbol{W}\boldsymbol{u} + \boldsymbol{b}), \tag{A.10}$$

where $\boldsymbol{\alpha}$ is the activation function.

A.2.3 Continuous ML models: NODE

As example for a continuous simulation model, we give the HUDA-ODE for a (shallow) NODE in the following. A NODE is obtained by combing a FFNN (to learn for the right-hand side) and an ODE solver to obtain a solution trajectory [11]. Compared to the FFNN (s. Eq. A.10) the major difference is that the FFNN is now evaluated for the function $\boldsymbol{f}$ (instead of $\boldsymbol{g}$), and takes the continuous state $\boldsymbol{x}_c$ as layer input (instead of the input $\boldsymbol{u}$):

$$\dot{\boldsymbol{x}}_c = \boldsymbol{f}(\boldsymbol{x}_c, \boldsymbol{p}) = \boldsymbol{\alpha}(\boldsymbol{W}\boldsymbol{x}_c + \boldsymbol{b}). \tag{A.11}$$

Of course, multiple FFNN layers can be chained to obtain a deep model for the right-hand side.

A.2.4 Discrete ML models: RNNs

As example for a discrete model, we give the HUDA-ODE of a RNN layer in the following. A RNN layer with m inputs and n outputs can be interpreted as fully discrete HUDA-ODEs that is sampled with a given frequency Δt. The observation that RNNs could be interpreted as forward Euler discretization of an ODE is also mentioned by Gholami *et al.* [109].

Parameters
The parameters of a RNN layer are given by the weights $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ and bias

$\boldsymbol{b} \in \mathbb{R}^n$, as well as the sampling frequency Δt:

$$\boldsymbol{p} = \begin{bmatrix} \Delta t \\ W_1^{(u)} \\ \vdots \\ W_{n \cdot m}^{(u)} \\ b_1^{(u)} \\ \vdots \\ b_n^{(u)} \\ W_1^{(x)} \\ \vdots \\ W_{n \cdot m}^{(x)} \\ b_1^{(x)} \\ \vdots \\ b_n^{(x)} \end{bmatrix}. \tag{A.12}$$

Discrete state
The discrete state $\boldsymbol{x}_d \in \mathbb{R}^{m+1}$ serves as memory of the last output. The first element, however, is used to store the next sampling time $\bar{t}^*$.

Output function
The output function is given by evaluation of the RNN layer. This involves multiplication of the input weights $\boldsymbol{W}^{(u)} \in \mathbb{R}^{m \times n}$ by the input $\boldsymbol{u} \in \mathbb{R}^m$, and multiplication of the state weights $\boldsymbol{W}^{(x)} \in \mathbb{R}^{m \times n}$ by the discrete state $\bar{\boldsymbol{x}}_d \in \mathbb{R}^m$

$$\boldsymbol{g}(\boldsymbol{x}_d, \boldsymbol{u}, \boldsymbol{p}) = \boldsymbol{\alpha}(\boldsymbol{W}_u \boldsymbol{u} + \boldsymbol{W}_x \cdot \bar{\boldsymbol{x}}_d + \boldsymbol{b}), \tag{A.13}$$

where

$$\bar{\boldsymbol{x}}_d = \begin{bmatrix} x_{d2} \\ \vdots \\ x_{dm} \end{bmatrix}, \tag{A.14}$$

so the discrete state excluding the first entry, that gives the next event time.

Event condition
The event condition is given by the distance to the next event time $\bar{t}^*$, that is part of the discrete state:

$$\boldsymbol{z} = \boldsymbol{c}(\boldsymbol{x}_d, t) = \left[\bar{t}^* - t\right]. \tag{A.15}$$

Event affect function
At an event instant, the discrete state $\boldsymbol{x}_d$ of the RNN is updated to match the previous outputs. The next event time is incremented.

$$\boldsymbol{x}_d(t^-) = \boldsymbol{x}(t^-) = \boldsymbol{a}(\boldsymbol{y}, \boldsymbol{p}, t^-) = \begin{bmatrix} t^- + \Delta t \\ \boldsymbol{y}(t^-) \end{bmatrix}. \tag{A.16}$$

Note, that the output function $\boldsymbol{y}$ could be replaced by the expressions in Eq. A.13 to make the affect function depending on the discrete state and input instead of the output.

A.2.5 Combined models: ODE-RNNs

As example for a combined model, we give the HUDA-ODE of the ODE-RNN in the following. ODE-RNNs are defined by the combination of a (neural) ODE and a RNN, together with an additional ANN for the output [65]. They are designed as alternative to RNNs, that can handle data sampled at irregular time steps. ODE-RNNs are used as recognition networks (as part of the encoder) in latent ODEs, as introduced in [65]. The functionality can be examined in Alg. 3.

Algorithm 3 The inference of an ODE-RNN for a given time series with n data points, as defined in [65]. Parameter and time function arguments are removed for simplicity, symbols are adapted.

Input: Data points and their timestamps $\{(\boldsymbol{u}_i, t_i)\}_{i=1..n}$
1: $\boldsymbol{h}_0 := 0$ ▷ the initial hidden state
2: **for** $i \in 1, 2, ..., n$ **do**
3: $\quad \boldsymbol{h}'_i := ODESolve(\boldsymbol{f}, \boldsymbol{h}_{i-1}, (t_{i-1}, t_i))$ ▷ integrating f from t_{i-1} to t_i
4: $\quad \boldsymbol{h}_i := RNNCell(\boldsymbol{h}'_i, \boldsymbol{u}_i)$ ▷ put integration result into the RNN
5: **end for**
6: $\boldsymbol{y}_i := OutputNN(\boldsymbol{h}_i)$ for all $i = 1..n$
Output: $\{\boldsymbol{y}_i\}_{i=1..n}; \boldsymbol{h}_n$

The core idea is straightforward: Instead of updating the recurrent cell based on the current input $\boldsymbol{u}_i$ and the *previous* hidden state $\boldsymbol{h}_i$, the hidden state is approximated by numerical integration of the function $\boldsymbol{f}$. Because numerical integration can be performed for any given time span (t_{i-1}, t_i), the ODE-RNN is capable of operating with irregular sampled inputs.

Here, $ODESolve$ corresponds to solving an ODE as used throughout this work, $RNNCell$ corresponds to the evaluation of a RNN and $OutputNN$ to the evaluation of the output FFNN. Of course, the important part of the algorithm is the inside of the time stepping loop:

$$\boldsymbol{h}'_i = ODESolve(\boldsymbol{f}, \boldsymbol{h}_{i-1}, (t_{i-1}, t_i)) \tag{A.17}$$

$$\boldsymbol{h}_i = RNNCell(\boldsymbol{h}'_i, \boldsymbol{u}_i). \tag{A.18}$$

Here, we refer to $\boldsymbol{h}'_i$ as *state approximation*, that is obtained by numerical integration and used as input to the RNN. The RNN updates the actual state $\boldsymbol{h}_i$ based on the approximated state $\boldsymbol{h}'_i$ and the input $\boldsymbol{u}_i$. This procedure corresponds to solving a discontinuous ODE, that jumps for every evaluation of the RNN.

Continuous ODE

The first line correspond to an ODE that is piecewise solved. This can be implemented by defining an ODE with condition and affect function, so that the correct state is updated for every discrete step. The right-hand side therefore states:

$$\dot{\boldsymbol{x}}_c = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{p}, t). \tag{A.19}$$

Event condition
As event condition, we can create an event indicator for every[1] given sample point:

$$\boldsymbol{c}(t) = \begin{bmatrix} t_1 - t \\ \vdots \\ t_n - t \end{bmatrix}. \tag{A.20}$$

Event affect
At any event instance, we evaluate the RNN and update the continuous state:

$$\boldsymbol{x}_c^+ = \boldsymbol{a}(\boldsymbol{x}_c^-, \boldsymbol{u}^-, \boldsymbol{p}, t^-) = RNNCell(\boldsymbol{x}_c^-, \boldsymbol{u}^-). \tag{A.21}$$

Output function
The output function can be defined straightforward and corresponds to the evaluation of a FFNN with input the state.

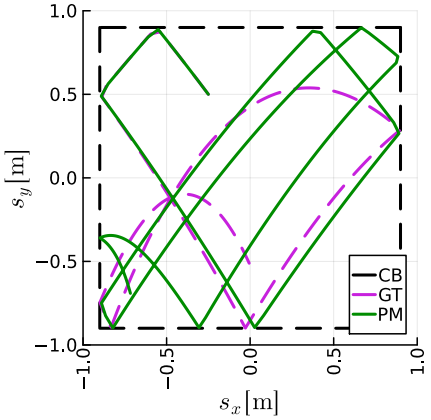
$$\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}_c^+, \boldsymbol{p}) = OutputNN(\boldsymbol{x}_c^+). \tag{A.22}$$

However, we note, that evaluation of the output network happens *after* event-handling, meaning the right state $\boldsymbol{x}_c^+$ is inserted, which corresponds to the RNN updated state.

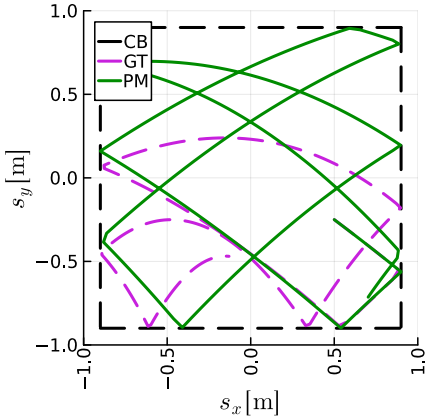
[1] More efficiently, we could generate a single event indicator and a discrete state (compare to the RNN in Sec. A.2.4) and step from one time event to the next. This makes this procedure more efficient, but also less readable.

A.3 Trajectories for the architecture experiment

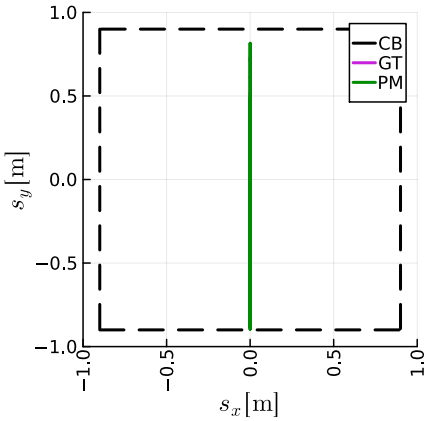
In Fig. A.2, we give the four training data trajectories for the architecture learning experiment. Trajectories are incorporated into the training data in the order of their numbering, for example, training on 3 trajectories involves the trajectories #1, #2, and #3.



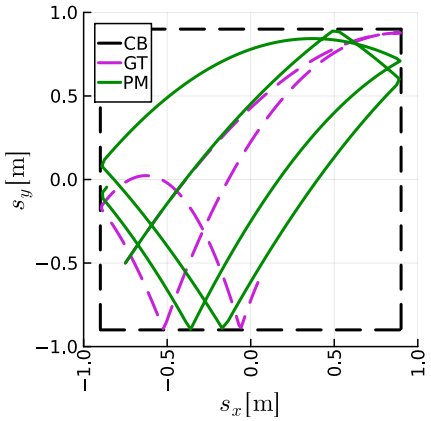
(a) Trajectory #1
$\boldsymbol{x}_0 = [-0.25, -6.0, 0.5, 8.0, 0]^T$



(b) Trajectory #2
$\boldsymbol{x}_0 = [0.5, 8.0, -0.25, -6.0, 0]^T$



(c) Trajectory #3
$\boldsymbol{x}_0 = [0.0, 0.0, 0.0, 4.0, 0]^T$



(d) Trajectory #4
$\boldsymbol{x}_0 = [-0.75, 4.0, -0.5, 6.0, 0]^T$

Figure A.2: The four training data trajectories for the architecture learning experiment. Plots show the collision border (CB), and the trajectories for ground truth (GT) and the physical model (PM) without air friction. Further, the initial state $\boldsymbol{x}_0$ is given in the captions.

A.4 Batch scheduling

As motivated, for large datasets batching is unavoidable. However, this also opens to the question on which batch element to pick next for an optimal training. In general, there are two (often) contradictory goals for the batch scheduler:

Fairness All elements should be included *equally* in the training, because there is in general no prior knowledge that states that same records are better or more valuable than others.

Prioritization *Important* elements should be sampled more often. The meaning of importance varies between different sampling strategies. For example, this could be valuable data that is underrepresented, or parts of the trajectory where the model performs unsatisfactorily.

In the following, different batch schedulers are briefly investigated, that where used for the ECM validation example, as well as for the JuliaCon 2023 workshop [97]. Nevertheless, this section serves only to complete the picture and actually deals with standard ML techniques.

Sequential

A very simple strategy that fulfills fairness by sequentially sampling all elements according to their chronological order. If several time series are part of the dataset, they are simply concatenated. For very heterogeneous datasets, this is obviously not an efficient strategy.

Random

In random sampling, a random element is drawn from the batch, with all elements having the same probability. This strategy is also unsuitable for heterogeneous data if the fastest possible training progress needs to be achieved. For small datasets, as we apply them within this thesis, it can be a valuable approach.

Max loss

To compensate for deficits in the appropriate choice of the next element, it can be an obvious strategy to sample the element with the biggest current error. There are two things to note here: If the states of the last step of the previous batch element are assumed as starting states, any error of the previous elements is also propagated into the chosen element, which leads to a falsification of the "largest error" assumption. Second, to determine the largest error, all batches must be updated for changes in the parameters, i.e. after each update step. Depending on the number of batches, this computational effort can be very high, so that alternative strategies must be developed. For example, only some elements can be updated and the resulting error (possibly only an element with a large error is sampled and not the largest error) is tolerated. Another problem is the "starvation" of batch elements with small errors, i.e. as long as elements feature large errors, elements with small errors are never sampled, which is equivalent to not using the corresponding training information. This can lead to problems in practice.

Loss accumulation

The *starvation* problem of the max loss scheduler can be alleviated by introducing error accumulation by adding up the errors over the update steps. Instead of the element with the largest error, the element with the largest *accumulated* error is now selected, and its accumulated error is then reset. This enables the sampling of elements with a large error at a higher rate, but at the same time prevents the

starvation of elements with a small error, because these also accumulate a large error with a sufficiently high number of update steps over the training process.

Auto-encoder-based latent space sampling (AEB-LSS)
Another approach is motivated by heterogeneous data, so data which is sampled very unevenly in state space. Obviously, picking random elements would lead towards a strong bias for learning data, that is present *more often*. However, this does not necessarily reflect the intended behavior. Often, and this is the core idea behind the proposed sampling technique, the goal would be to have *fair* sampling over the entire state space[2] of the system. Technically random samples from state space can be drawn by discretizing the state space into a finite number of volumes and implement a two stage approach, where first one of the volumes is selected, and second, an element within the volume is picked. The choice of volume can be random, or more sophisticated sampling techniques like Latin hypercube sampling or orthogonal sampling can be applied.

For the general case of systems including inputs and physical parameters that may vary, these should also be included to the sampling space. However, the dimension for sampling can rapidly grow and not every state, input and parameter has a significant impact on the system. To automatically select a meaningful subspace for sampling, an auto-encoder can be applied. By minimizing the reconstruction error, the auto-encoder learns for a latent (but smaller) subspace that has a strong bias towards the relevant inputs, states and parameters. This technique is referred to as *latent space sampling* and is a common approach in ML. For larger data spaces, also variational auto-encoderss (VAEs) and derived models are applied.

This additional procedure is motivated by the validation example in Sec. 8.3, that features a huge variance of multiple orders of magnitude between samples at different locations in state space. At the same time, the model features 4 states and 32 inputs, whereas especially the inputs are known to not have an equally distributed impact on the system.

A.5 Nested AD in eigen-informed loss functions

The source for nested AD in eigen-informed regularization is briefly investigated by examination of the cost function incorporating eigenvalue determination:

$$\boldsymbol{l}_\lambda(\boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}),\boldsymbol{p})). \tag{A.23}$$

It is important to note, that the eigenvalue computation $\boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}),\boldsymbol{p})$ depends implicitly on the parameters $\boldsymbol{p}$ (via the solution points to determine the linearization for), however, also *explicitly* because the right-hand side is evaluated for the linearization. As a result, computing the parameter gradient results in:

$$\frac{\partial \boldsymbol{l}_\lambda(\boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}),\boldsymbol{p}))}{\partial \boldsymbol{p}} = \frac{\partial \boldsymbol{l}_\lambda(\boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}),\boldsymbol{p}))}{\partial \boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}))} \cdot \left(\frac{\partial \boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}),\boldsymbol{p})}{\partial \boldsymbol{X}(\boldsymbol{p})} \cdot \frac{\partial \boldsymbol{X}(\boldsymbol{p})}{\partial \boldsymbol{p}} + \frac{\partial \boldsymbol{\lambda}(\boldsymbol{X}(\boldsymbol{p}),\boldsymbol{p})}{\partial \boldsymbol{p}} \right). \tag{A.24}$$

By comparing to the loss function defined on the ODE solution (s. Eq. 2.21), we find that the Jacobian $\partial \boldsymbol{X}/\partial \boldsymbol{p}$ is required for SA including eigenvalues, as well as for

[2] Or *operation space*, if only a part of the state space is relevant for the later application of the model.

SA including the ODE solution, and can therefore be shared for loss functions that incorporate the ODE solution *and* eigenvalue regularization. Further we find, that the Jacobian $\partial \boldsymbol{l}_\lambda / \partial \boldsymbol{\lambda}$ is computational cheap, because it does only require differentiation of the algebraically simple cost functions for eigenvalues that we introduced, compare Eq. 6.12-6.23. The last remaining Jacobians $\partial \boldsymbol{\lambda} / \partial \boldsymbol{X}$ and $\partial \boldsymbol{\lambda} / \partial \boldsymbol{p}$ are identified the expensive part of the computation, as we derive now.

The Jacobian $\partial \boldsymbol{\lambda} / \partial \boldsymbol{X}$ corresponds to the operation of computing eigenvalues for a given ODE solution, which actually involves two steps: First, the system is linearized $\boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p})$ for every point of the solution, and second, the eigenvalues of the linearization are computed $\boldsymbol{\lambda}(\boldsymbol{A})$. The Jacobians can be investigated in more detail:

$$\frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{X}} = \frac{\partial \boldsymbol{\lambda}(\boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p}))}{\partial \boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p})} \frac{\partial \boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p})}{\partial \boldsymbol{X}} \tag{A.25}$$

and

$$\frac{\partial \boldsymbol{\lambda}}{\partial \boldsymbol{p}} = \frac{\partial \boldsymbol{\lambda}(\boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p}))}{\partial \boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p})} \frac{\partial \boldsymbol{A}(\boldsymbol{X}, \boldsymbol{p})}{\partial \boldsymbol{p}}. \tag{A.26}$$

Keeping in mind that $\boldsymbol{A}$ corresponds to linearization, therefore $\partial \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{p}) / \partial \boldsymbol{x}$, we can substitute $\boldsymbol{A}$ in the final expressions and find the cause for nested AD.

A.6 Hyperparameters of the SCARA experiment

Table A.1: Hyperparameters, search space, and search results for the SCARA example.

Hyperparameter	Space	Result
η	$[10^{-2}, 10^{-4}]$	10^{-3}
β_1	$\{0.99, 0.9\}$	0.9
β_2	$\{0.9999, 0.999, 0.99\}$	0.999
batch duration	$[0.01\,\mathrm{s}, 0.1\,\mathrm{s}]$	$0.05\,\mathrm{s}$
loss function	{MAE, MSE}	MAE
layer count	$\{2, 3, 4\}$	3
layer width	$\{8, 16, 32\}$	32
gate opening (ANN)	$[0.0, 0.5]$	0.2

A.7 Hyperparameters of the ECM experiment

Table A.2: Hyperparameters, search space, and search results for the ECM example.

Hyperparameter	Space	Result
η	$[10^{-4}, 10^{-6}]$	$3 \cdot 10^{-6}$
β_1	$\{0.99, 0.9\}$	0.9
β_2	$\{0.9999, 0.999, 0.99\}$	0.99
batch duration	$[0.01\,\mathrm{s}, 0.1\,\mathrm{s}]$	0.08 s
last weight	$\{0.0, 0.1, 0.2\}$	0.0
scheduler	{random, loss accumulation, AEB-LSS} (s. Sec. A.4)	random
loss function	{MAE, MSE}	MAE
layer count	$\{2, 3, 4\}$	3
layer width	$\{8, 16, 32\}$	8
loss scaling ratio	$[0.0, 1.0]$	0.7
gate opening (ANN)	$[0.0, 0.5]$	0.4
activation	{tanh, relu, leakyrelu}	tanh

A.8 Hyperparameters of the PCS experiment

Table A.3: Hyperparameters, search space, and search results for the PCS example.

Hyperparameter	Space	Result
η	$[10^{-5}, 10^{-3}]$	$32 \cdot 10^{-5}$
β_1	$\{0.99, 0.9\}$	0.9
β_2	$\{0.9999, 0.999, 0.99\}$	0.99
batch duration	$[0.05\,\mathrm{s}, 0.15\,\mathrm{s}]$	0.14 s
last weight	$\{0.0, 0.1, 0.2\}$	0.1
loss portion	$[0.1, 1.0]$	0.8
loss function	{MAE, MSE}	MAE
layer count	$\{2, 3, 4\}$	3
layer width	$\{16, 32, 64, 128\}$	128
gate opening (ANN)	$[10^{-2}, 10^{-1}]$	10^{-2}
activation	{tanh, sigmoid}	tanh
architecture	{Arteries, All}	Arteries
grad. clipping	$[10^{1}, 10^{3}]$	10^{3}

A.9 Test results for the SCARA experiment

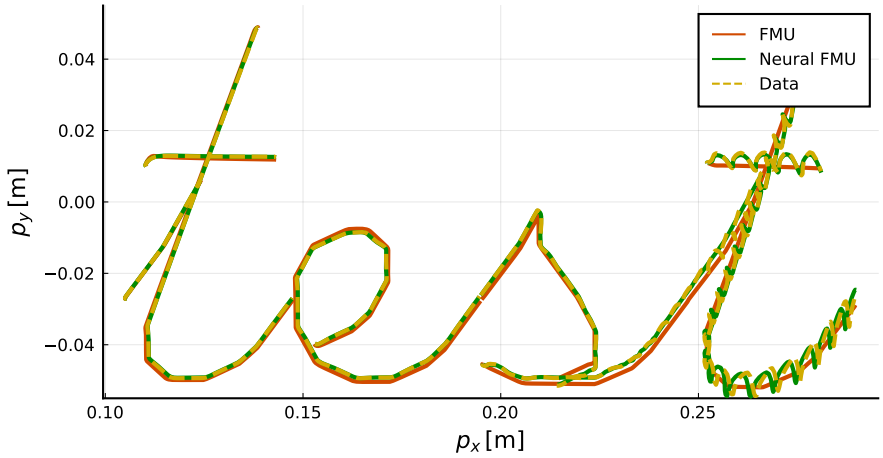
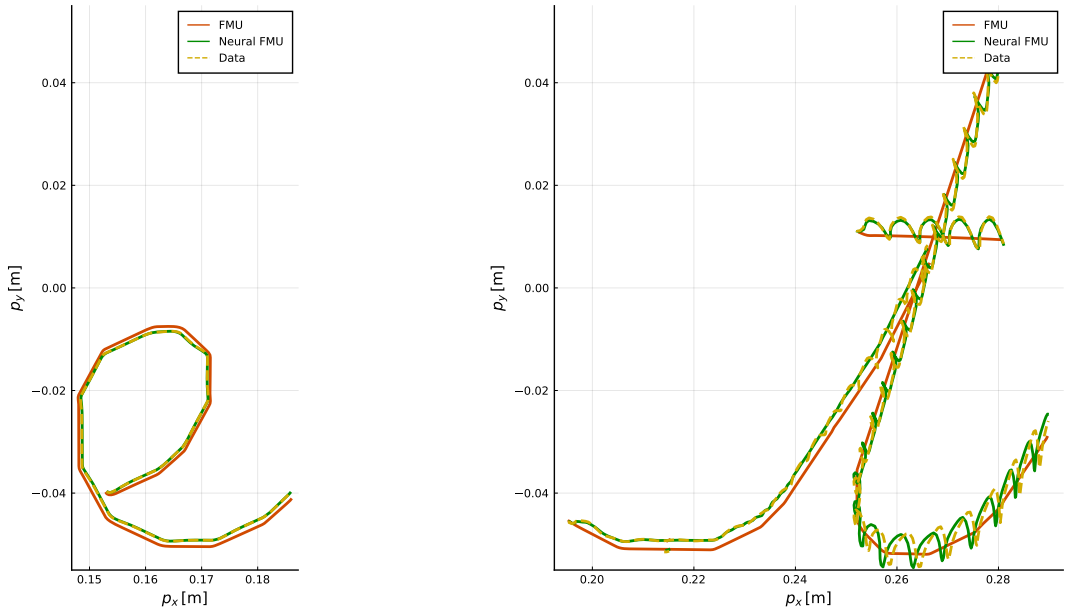


Figure A.3: Results for the SCARA experiment, comparing FMU and neural FMU performance on the word “test”.



(a) The letter “e”.

(b) The letter “t”.

Figure A.4: Results for the SCARA experiment, comparing FMU and neural FMU performance on the letters “e” and “t” (the rear one) in the word “test”.

A.10 Validation results for the ECM experiment

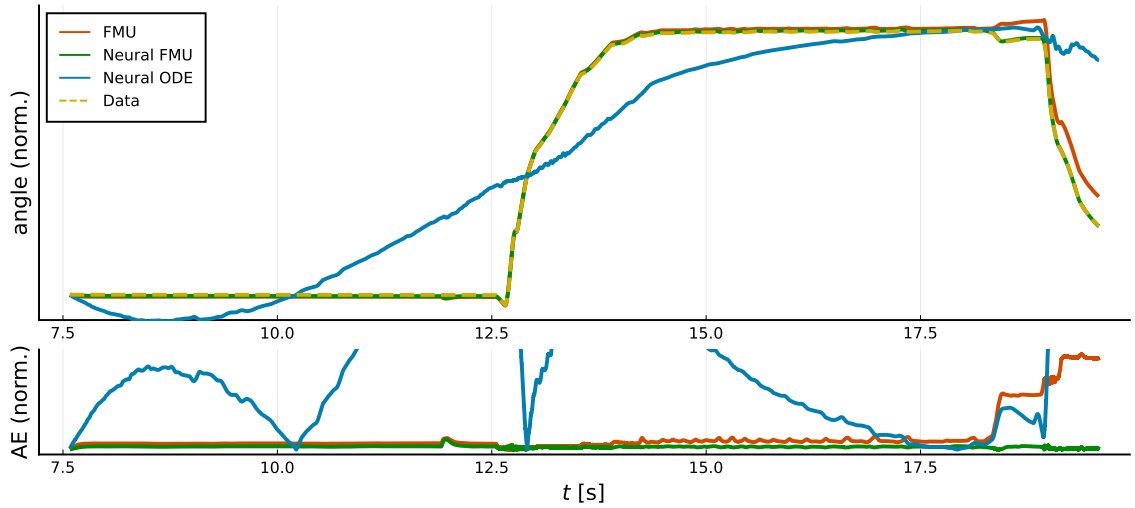


Figure A.5: The ECM rotor angle on the validation scenario, comparing the physical (FMU), hybrid (neural FMU), and ML (neural ODE) model. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

Table A.4: Comparison of the solver steps (*stiffness*) and the MAE for the different models on validation data. Errors are normalized for reasons of IP.

Model	MAE (angle)	MAE (speed)	Acc. steps	Rej. steps
FMU (physical)	0.0131	0.0085	269,556	31,865
Neural FMU (hybrid)	0.0028	0.0043	128,593	3,416
Neural ODE (ML)	0.1063	0.1369	221	16

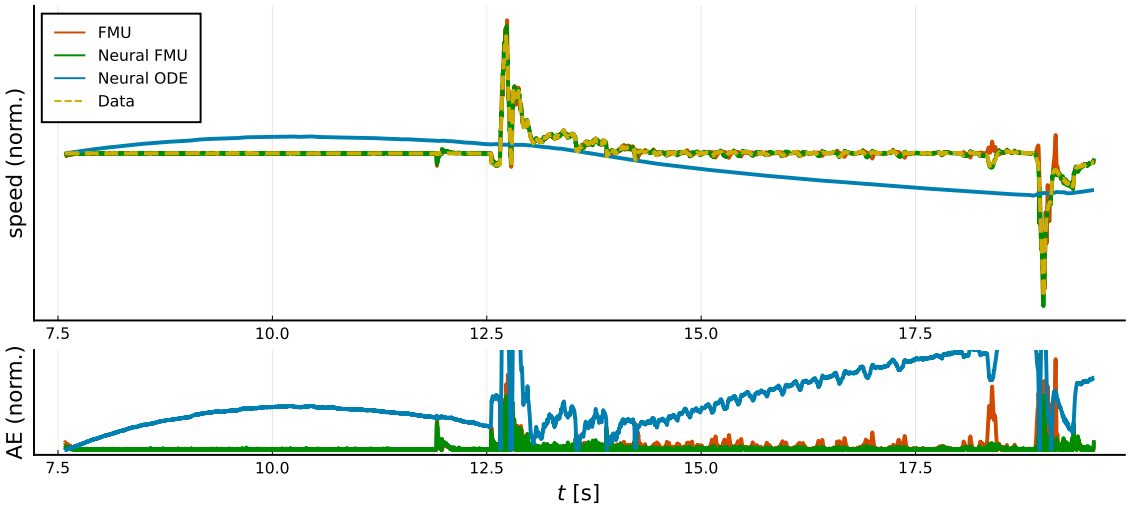


Figure A.6: The ECM rotor speed on the validation scenario, comparing the physical (FMU), hybrid (neural FMU), and ML (neural ODE) model. The upper plot shows the simulation trajectories, the lower the absolute deviation between the corresponding model and data over time. The y-axis is normalized, axis labels are removed for reasons of IP, and the axis limits are clipped to ignore the neural ODE.

A.11 Validation results for the PCS experiment

Table A.5: Comparison of the inference times, solver steps (*stiffness*), and MAE for the different models on validation data. Errors are normalized for reasons of IP.

Model	Sim. time [s]	MAE (vol. flow)	MAE (area)	Acc. steps	Rej. steps
FMU (fine-grid, phy.)	503.63	0.0	0.0	9,868	958
FMU (coarse-grid, phy.)	0.092	1.3051	0.2997	290	17
Neural FMU (hybrid)	1.001	0.0666	0.0511	184	30
Neural ODE (ML)	0.148	5906.3371	33436.2086	101	0

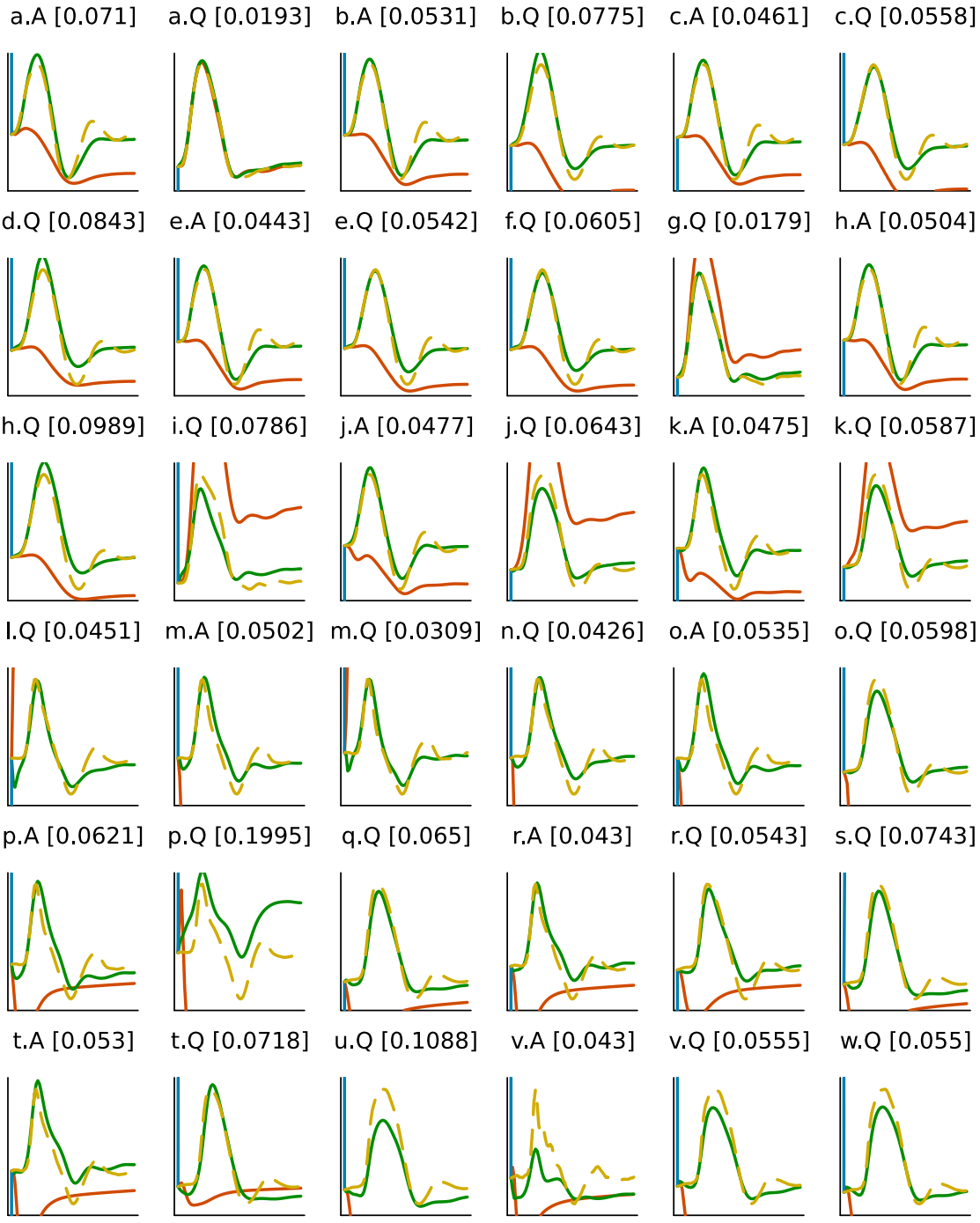


Figure A.7: Validation results of the PCS experiment. The plots show the arterial cross-section area (A) and volume flow rate (Q) for the 23 arterial sections (a-w) on the y-axis over the time of a single heart beat on the x-axis. The fine-grid model (yellow, dashed) is compared to the coarse-grid model (red), the ML model (blue, clipped), and the hybrid model (green). The subplot titles identify the arterial section and quantity, and give the MAE for the corresponding quantity. All quantities are normalized for IP-reasons.

References

[1] "The bitter lesson," Accessed: Sep. 6, 2023. [Online]. Available: http://incompleteideas.net/IncIdeas/BitterLesson.html.

[2] T. Sauerer, G. F. Velázquez, and C. Schmid, *Relapse of acute myeloid leukemia after allogeneic stem cell transplantation: Immune escape mechanisms and current implications for therapy*, 2023. DOI: https://doi.org/10.1186/s12943-023-01889-6.

[3] L. Rehberg and A. Brem, "Industrial prototyping in the german automotive industry: Bridging the gap between physical and virtual prototypes," *Journal of Engineering and Technology Management*, vol. 71, p. 101 798, 2024, ISSN: 0923-4748. DOI: https://doi.org/10.1016/j.jengtecman.2024.101798. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0923474824000031.

[4] Y. Lu *et al.*, *Machine learning for synthetic data generation: A review*, 2025. arXiv: 2302.04062 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2302.04062.

[5] R. Bommasani *et al.*, *On the opportunities and risks of foundation models*, 2022. arXiv: 2108.07258 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2108.07258.

[6] P. Chlap, H. Min, N. Vandenberg, J. Dowling, L. Holloway, and A. Haworth, "A review of medical image data augmentation techniques for deep learning applications," *Journal of Medical Imaging and Radiation Oncology*, vol. 65, no. 5, pp. 545–563, 2021. DOI: https://doi.org/10.1111/1754-9485.13261. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1754-9485.13261. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/1754-9485.13261.

[7] J. Sansana *et al.*, "Recent trends on hybrid modeling for industry 4.0," *Computers & Chemical Engineering*, vol. 151, p. 107 365, 2021, ISSN: 0098-1354. DOI: https://doi.org/10.1016/j.compchemeng.2021.107365. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0098135421001435.

[8] N. Baker *et al.*, "Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence," Feb. 2019. DOI: 10.2172/1478744. [Online]. Available: https://www.osti.gov/biblio/1478744.

[9] D. C. Psichogios and L. H. Ungar, "A hybrid neural network-first principles approach to process modeling," *AIChE Journal*, vol. 38, no. 10, pp. 1499–1511, 1992. DOI: https://doi.org/10.1002/aic.690381003. eprint: https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.690381003. [Online]. Available: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690381003.

[10] "What is scientific machine learning?" Accessed: Sep. 6, 2023. [Online]. Available: https://sites.brown.edu/bergen-lab/research/what-is-sciml/.

[11] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," *CoRR*, vol. abs/1806.07366, 2018. arXiv: 1806.07366. [Online]. Available: http://arxiv.org/abs/1806.07366.

[12] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019, ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[13] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, *ModelingToolkit: A composable graph transformation system for equation-based modeling*, 2021. arXiv: 2103.05244 [cs.MS].

[14] C. Rackauckas and Q. Nie, "DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in Julia," *The Journal of Open Research Software*, vol. 5, no. 1, 2017, Exported from https://app.dimensions.ai on 2019/05/05. DOI: 10.5334/jors.151. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1085583166%20and%20http://openresearchsoftware.metajnl.com/articles/10.5334/jors.151/galley/245/download/.

[15] F. E. Cellier and E. Kofman, *Continuous system simulation*. Springer Science & Business Media, 2006.

[16] R. T. Q. Chen, B. Amos, and M. Nickel, "Learning neural event functions for ordinary differential equations," *CoRR*, vol. abs/2011.03902, 2020. arXiv: 2011.03902. [Online]. Available: https://arxiv.org/abs/2011.03902.

[17] C. C. Pantelides, "The consistent initialization of differential-algebraic systems," *SIAM Journal on scientific and statistical computing*, vol. 9, no. 2, pp. 213–231, 1988.

[18] S. Ruder, *An overview of gradient descent optimization algorithms*, 2017. arXiv: 1609.04747 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1609.04747.

[19] Y. Ma, V. Dixit, M. J. Innes, X. Guo, and C. Rackauckas, "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2021, pp. 1–9.

[20] F. Sapienza *et al.*, *Differentiable programming for differential equations: A review*, 2024. arXiv: 2406.09699 [math.NA]. [Online]. Available: https://arxiv.org/abs/2406.09699.

[21] A. Pfeiffer, "Numerische sensitivitätsanalyse unstetiger multidisziplinärer modelle mit anwendungen in der gradientenbasierten optimierung," Ph.D. dissertation, Martin-Luther Universität Halle-Wittenberg, 2008.

[22] H. Zhang and A. Sandu, "FATODE: A library for forward, adjoint, and tangent linear integration of ODEs," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C504–C523, 2014. DOI: 10.1137/130912335. eprint: https://doi.org/10.1137/130912335. [Online]. Available: https://doi.org/10.1137/130912335.

[23] D. Onken and L. Ruthotto, *Discretize-optimize vs. optimize-discretize for time-series regression and continuous normalizing flows*, 2020. arXiv: 2005.13420 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2005.13420.

[24] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991, ISSN: 0360-0300. DOI: 10.1145/103162.103163. [Online]. Available: https://doi.org/10.1145/103162.103163.

[25] Z. Ma *et al.*, "BaGuaLu: Targeting brain scale pretrained models with over 37 million cores," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 192–204, ISBN: 9781450392044. DOI: 10.1145/3503221.3508417. [Online]. Available: https://doi.org/10.1145/3503221.3508417.

[26] A. C. Hindmarsh *et al.*, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 363–396, 2005.

[27] R. Serban and A. C. Hindmarsh, "CVODES: An ODE solver with sensitivity analysis capabilities," Citeseer, Tech. Rep., 2003.

[28] D. Hilbert and R. Courant, "Methoden der mathematischen Physik," 1924.

[29] S. Abhyankar, M. Anitescu, E. Constantinescu, and H. Zhang, "Efficient adjoint computation of hybrid systems of differential algebraic equations with applications in power systems," Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2016.

[30] H. Zhang, S. Abhyankar, E. Constantinescu, and M. Anitescu, "Discrete adjoint sensitivity analysis of hybrid dynamical systems with switching," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 5, pp. 1247–1259, 2017.

[31] D. E. Stewart and M. Anitescu, "Optimal control of systems with discontinuous differential equations," *Numerische Mathematik*, vol. 114, no. 4, pp. 653–695, 2010.

[32] S. Corner, C. Sandu, and A. Sandu, *Adjoint sensitivity analysis of hybrid multibody dynamical systems*, 2018. arXiv: 1802.07188 [math.OC]. [Online]. Available: https://arxiv.org/abs/1802.07188.

[33] B. J. Driessen and N. Sadegh, "On the discontinuity of the costates for optimal control problems with coulomb friction," *Optimal Control Applications and Methods*, vol. 22, no. 4, pp. 197–200, 2001.

[34] T.-H. Kim and I.-J. Ha, "Time-optimal control of a single-DOF mechanical system with friction," *IEEE Transactions on Automatic Control*, vol. 46, no. 5, pp. 751–755, 2001.

[35] W. Zhu, K. Xu, E. Darve, and G. C. Beroza, "A general approach to seismic inversion with automatic differentiation," *Computers & Geosciences*, vol. 151, p. 104 751, 2021, ISSN: 0098-3004. DOI: https://doi.org/10.1016/j.cageo.2021.104751. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0098300421000595.

[36] C. Rackauckas *et al.*, *Universal differential equations for scientific machine learning*, 2021. arXiv: 2001.04385 [cs.LG].

[37] S. Kim, W. Ji, S. Deng, Y. Ma, and C. Rackauckas, "Stiff neural ordinary differential equations," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 31, no. 9, 2021.

[38] J. Carson, "Introduction to modeling and simulation," in *Proceedings of the Winter Simulation Conference, 2005.*, 2005, pp. 16–23. DOI: 10.1109/WSC.2005.1574235.

[39] L. G. Birta and G. Arbez, *Modelling and simulation, Exploring Dynamic System Behaviour*. Springer Cham, 2013. DOI: https://doi.org/10.1007/978-3-030-18869-6.

[40] U. H. E. Center, "HEC-HMS technical reference manual," USACE Hydrologic Engineering Center, ..., Tech. Rep., May 2024. [Online]. Available: https://github.com/modelica/fmi-standard/releases/download/v2.0.2/FMI-Specification-2.0.2.pdf.

[41] Modelica Association, "Functional mock-up interface for model exchange and co-simulation. document version: 2.0.2," Modelica Association, Linköping, Tech. Rep., Dec. 2020. [Online]. Available: https://github.com/modelica/fmi-standard/releases/download/v2.0.2/FMI-Specification-2.0.2.pdf.

[42] "Homepage of FMI standard," Accessed: Sep. 6, 2023. [Online]. Available: https://fmi-standard.org/.

[43] Modelica Association, "Functional mock-up interface specification. version 3.0.1," Modelica Association, Tech. Rep., Jul. 2023. [Online]. Available: https://fmi-standard.org/docs/3.0.1/.

[44] J. Bezanson, S. Karpinsky, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *CoRR*, vol. abs/1209.5145, 2012. arXiv: 1209.5145. [Online]. Available: http://arxiv.org/abs/1209.5145.

[45] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *CoRR*, vol. abs/1411.1607, 2015. arXiv: 1411.1607. [Online]. Available: http://arxiv.org/abs/1411.1607.

[46] A. Claster. "Newsletter august 2023: Julia enters TIOBE index top 20, JuliaCon highlights and more," Accessed: Dec. 28, 2024. [Online]. Available: https://info.juliahub.com/blog/newsletter-august-2023-julia-enters-tiobe-index-top-20-juliacon-highlights-and-more.

[47] A. Pal, *Lux: Explicit parameterization of deep neural networks in Julia*, version v1.4.2, If you use this software, please cite it as below., Apr. 2023. DOI: 10.5281/zenodo.7808903. [Online]. Available: https://doi.org/10.5281/zenodo.7808903.

[48] A. Pal, "On Efficient Training & Inference of Neural Differential Equations," Massachusetts Institute of Technology, 2023.

[49] M. Innes *et al.*, "Fashionable modelling with flux," *CoRR*, vol. abs/1811.01457, 2018. arXiv: 1811.01457. [Online]. Available: https://arxiv.org/abs/1811.01457.

[50] M. Innes, "Flux: Elegant machine learning with Julia," *Journal of Open Source Software*, 2018. DOI: 10.21105/joss.00602.

[51] W. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 12 472–12 485. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf.

[52] W. S. Moses *et al.*, "Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21, St. Louis, Missouri: Association for Computing Machinery, 2021, ISBN: 9781450384421. DOI: 10.1145/3458817.3476165. [Online]. Available: https://doi.org/10.1145/3458817.3476165.

[53] M. Innes, "Don't unroll adjoint: Differentiating SSA-form programs," *CoRR*, vol. abs/1810.07951, 2018. arXiv: 1810.07951. [Online]. Available: http://arxiv.org/abs/1810.07951.

[54] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016. [Online]. Available: https://arxiv.org/abs/1607.07892.

[55] T. A. Johansen and B. A. Foss, "Representing and learning unmodeled dynamics with neural network memories," in *1992 American Control Conference*, 1992. DOI: 10.23919/ACC.1992.4792705.

[56] J. Schubert, R. Simutis, M. Dors, I. Havlík, and A. Lübbert, "Hybrid modelling of yeast production processes – combination of a priori knowledge on different levels of sophistication," *Chemical Engineering & Technology*, vol. 17, no. 1, pp. 10–20, 1994. DOI: https://doi.org/10.1002/ceat.270170103. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ceat.270170103. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ceat.270170103.

[57] B. Duarte, P. M. Saraiva, and C. C. Pantelides, "Combined mechanistic and empirical modelling," *International Journal of Chemical Reactor Engineering*, vol. 2, no. 1, 2004. DOI: doi:10.2202/1542-6580.1128. [Online]. Available: https://doi.org/10.2202/1542-6580.1128.

[58] Y. Oussar and G. Dreyfus, "How to be a gray box: Dynamic semi-physical modeling," *Neural Networks*, vol. 14, no. 9, pp. 1161–1172, 2001, ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(01)00096-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608001000096X.

[59] H. Braake, H. van Can, and H. Verbruggen, "Semi-mechanistic modeling of chemical processes with neural networks," *Engineering Applications of Artificial Intelligence*, vol. 11, no. 4, pp. 507–515, 1998, ISSN: 0952-1976. DOI: https://doi.org/10.1016/S0952-1976(98)00011-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197698000116.

[60] "Neuro-mechanistic modeling," Accessed: Aug. 27, 2024. [Online]. Available: https://www.dfki.de/en/web/research/research-departments/neuro-mechanistic-modeling.

[61] H.-T. Su, N. Bhat, P. Minderman, and T. McAvoy, "Integrating neural networks with first principles models for dynamic modeling," *IFAC Proceedings Volumes*, vol. 25, no. 5, pp. 327–332, 1992, 3rd IFAC Symposium on Dynamics and Control of Chemical Reactors, Distillation Columns and Batch Processes (DYCORD+ '92), Maryland, USA, 26-29 April, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)51013-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017510137.

[62] A. Choudhary, J. F. Lindner, E. G. Holliday, S. T. Miller, S. Sinha, and W. L. Ditto, "Physics-enhanced neural networks learn order and chaos," *Phys. Rev. E*, vol. 101, p. 062 207, 6 Jun. 2020. DOI: 10.1103/PhysRevE.101.062207. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.101.062207.

[63] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021.

[64] N. Thuerey, P. Holl, M. Müller, P. Schnell, F. Trost, and K. Um, "Physics-based deep learning," *CoRR*, vol. abs/2109.05237, 2021. arXiv: 2109.05237. [Online]. Available: https://arxiv.org/abs/2109.05237.

[65] Y. Rubanova, R. T. Q. Chen, and D. Duvenaud, *Latent ODEs for irregularly-sampled time series*, 2019. arXiv: 1907.03907 [cs.LG].

[66] Q. Cao, S. Liu, A. J. Varghese, J. Darbon, M. Triantafyllou, and G. E. Karniadakis, *Automatic selection of the best neural architecture for time series forecasting via multi-objective optimization and pareto optimality conditions*, 2025. arXiv: 2501.12215 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2501.12215.

[67] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[68] K. Cho *et al.*, *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, 2014. arXiv: 1406.1078 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1406.1078.

[69] A. Vaswani *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[70] S. Zendehboudi, N. Rezaei, and A. Lohi, "Applications of hybrid models in chemical, petroleum, and energy systems: A systematic review," *Applied Energy*, vol. 228, pp. 2539–2566, 2018, ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2018.06.051. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261918309218.

[71] S. B. Padmanabhan, M. T. Mabrouk, and B. Lacarrière, "A hybrid model for packed bed thermal energy storage system," *Journal of Energy Storage*, vol. 98, p. 113 068, 2024, ISSN: 2352-152X. DOI: https://doi.org/10.1016/j.est.2024.113068. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352152X24026549.

[72] S. B. Padmanabhan, M. T. Mabrouk, and B. Lacarrière, "Neural-Accelerated Dynamic modeling of heat pumps," *Applied Thermal Engineering*, vol. 274, p. 126 653, 2025. DOI: 10.1016/j.applthermaleng.2025.126653. [Online]. Available: https://hal.science/hal-05086722.

[73] J. A. Kuzhiyil, T. Damoulas, and W. D. Widanage, "Neural equivalent circuit models: Universal differential equations for battery modelling," *Applied Energy*, vol. 371, p. 123 692, 2024, ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2024.123692. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306261924010754.

[74] D. Rodrigue, M. T. Mabrouk, B. Pasdeloup, P. Meyer, and B. Lacarrière, "Leveraging Neural Networks in a Hybrid Model Predictive control framework for district heating networks," in *Proceedings of the 37th International Conference on Efficiency, Cost, Optimization, Simulation and Environmental Impact of Energy Systems (ECOS 2024)*, Rhodes (Rodos) Island, Greece, Jun. 2024, pp. 287–298. DOI: 10.52202/077185-0025. [Online]. Available: https://hal.science/hal-04611500.

[75] L. von Rueden, S. Mayer, R. Sifa, C. Bauckhage, and J. Garcke, "Combining machine learning and simulation to a hybrid modelling approach: Current and future directions," in *Advances in Intelligent Data Analysis XVIII*, M. R. Berthold, A. Feelders, and G. Krempl, Eds., Cham: Springer International Publishing, 2020, pp. 548–560.

[76] W. H. Joerding and J. L. Meador, "Encoding a priori information in feedforward networks," *Neural Networks*, vol. 4, no. 6, pp. 847–856, 1991, ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(91)90063-B. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089360809190063B.

[77] M. L. Thompson and M. A. Kramer, "Modeling chemical processes using prior knowledge and neural networks," *AIChE Journal*, vol. 40, no. 8, pp. 1328–1340, 1994. DOI: https://doi.org/10.1002/aic.690400806. eprint: https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.690400806. [Online]. Available: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690400806.

[78] M. Rudolph, S. Kurz, and B. Rakitsch, "Hybrid modeling design patterns," *Journal of Mathematics in Industry*, vol. 14, no. 1, pp. 1–36, 2024.

[79] B. Sohlberg and E. Jacobsen, "Grey box modelling – branches and experiences," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 11 415–11 420, 2008, 17th IFAC World Congress, ISSN: 1474-6670. DOI: https://doi.org/10.3182/20080706-5-KR-1001.01934. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016408025.

[80] F. Jirasek and H. Hasse, "Combining machine learning with physical knowledge in thermodynamic modeling of fluid mixtures," *Annual Review of Chemical and Biomolecular Engineering*, vol. 14, no. Volume 14, 2023, pp. 31–51, 2023, ISSN: 1947-5446. DOI: https://doi.org/10.1146/annurev-chembioeng-092220-025342. [Online]. Available: https://www.annualreviews.org/content/journals/10.1146/annurev-chembioeng-092220-025342.

[81] L. P. Leifsson, H. Saevarsdóttir, S. P. Sigurosson, and A. Vésteinsson, "Grey-box modeling of an ocean vessel for operational optimization," *Simulation Modelling Practice and Theory*, vol. 16, no. 8, pp. 923–932, 2008, EUROSIM 2007, ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2008.03.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X08000488.

[82] Z. Lai, C. Mylonas, S. Nagarajaiah, and E. Chatzi, "Structural identification with physics-informed neural ordinary differential equations," *Journal of Sound and Vibration*, vol. 508, p. 116 196, 2021, ISSN: 0022-460X. DOI: https://doi.org/10.1016/j.jsv.2021.116196. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022460X21002686.

[83] T. Thummerer, J. Stoljar, and L. Mikelsons, "NeuralFMU: Presenting a workflow for integrating hybrid NeuralODEs into real-world applications," *Electronics*, vol. 11, no. 19, p. 3202, 2022. DOI: 10.3390/electronics11193202. [Online]. Available: https://www.mdpi.com/2079-9292/11/19/3202.

[84] T. Thummerer and L. Mikelsons, *Learnable & interpretable model combination in dynamic systems modeling*, 2024. arXiv: 2406.08093 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2406.08093.

[85] L. A. Ochel and B. Bachmann, "Initialization of equation-based hybrid models within OpenModelica," in *EOOLT*, 2013, pp. 97–103.

[86] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. DOI: 10.1137/0201010. eprint: https://doi.org/10.1137/0201010. [Online]. Available: https://doi.org/10.1137/0201010.

[87] V. Waurich, I. Gubsch, C. Schubert, and M. Walther, "Reshuffling: A symbolic pre-processing algorithm for improved robustness, performance and parallelization for the simulation of differential algebraic equations," in *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, 2014, pp. 3–10.

[88] A. Norcliffe, C. Bodnar, B. Day, N. Simidjievski, and P. Lió, "On second order behaviour in augmented neural ODEs," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 5911–5921. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/418db2ea5d227a9ea8db8e5357ca2084-Paper.pdf.

[89] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html.

[90] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[91] S. K. Kumar, *On weight initialization in deep neural networks*, 2017. arXiv: 1704.08863 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1704.08863.

[92] T. Thummerer, J. Tintenherr, and L. Mikelsons, "Hybrid modeling of the human cardiovascular system using NeuralFMUs," *Journal of Physics: Conference Series*, vol. 2090, no. 1, p. 012 155, 2021. DOI: 10.1088/1742-6596/2090/1/012155.

[93] T. Thummerer, A. Kolesnikov, J. Gundermann, D. Ritz, and L. Mikelsons, "Paving the way for hybrid twins using neural functional mock-up units," in *Proceedings of the 15th International Modelica Conference 2023, Aachen, October 9-11*, D. Müller, A. Monti, and A. Benigni, Eds., 2023, pp. 141–150. DOI: 10.3384/ecp204141.

[94] L. T. Biegler, *Nonlinear programming: concepts, algorithms, and applications to chemical processes*. SIAM, 2010.

[95] E. M. Turan and J. Jäschke, "Multiple shooting for training neural differential equations on time series," *IEEE Control Systems Letters*, vol. 6, pp. 1897–1902, 2021.

[96] F. Bruder and L. Mikelsons, "Modia and Julia for grey box modeling," in *Modelica Conferences*, 2021, pp. 87–95.

[97] T. Thummerer and L. Mikelsons. "Using NeuralODEs in real life applications," Accessed: Feb. 7, 2025. [Online]. Available: https://pretalx.com/juliacon2023/talk/EWL3LC/.

[98] D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[99] H. Liang and H. Wu, "Parameter estimation for differential equation models using a framework of measurement error in regression models," *Journal of the American Statistical Association*, vol. 103, no. 484, pp. 1570–1583, 2008, PMID: 19956350. DOI: 10.1198/016214508000000797. eprint: https://doi.org/10.1198/016214508000000797. [Online]. Available: https://doi.org/10.1198/016214508000000797.

[100] E. Roesch, C. Rackauckas, and M. P. H. Stumpf, "Collocation based training of neural ordinary differential equations," *Statistical Applications in Genetics and Molecular Biology*, vol. 20, no. 2, pp. 37–49, 2021. DOI: doi:10.1515/sagmb-2020-0025. [Online]. Available: https://doi.org/10.1515/sagmb-2020-0025.

[101] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[102] N. Rahaman *et al.*, "On the spectral bias of neural networks," in *International conference on machine learning*, PMLR, 2019, pp. 5301–5310.

[103] Z.-Q. J. Xu, Y. Zhang, and Y. Xiao, "Training behavior of deep neural network in frequency domain," in *Neural Information Processing: 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12–15, 2019, Proceedings, Part I 26*, Springer, 2019, pp. 264–274.

[104] E. Dupont, A. Doucet, and Y. W. Teh, *Augmented neural ODEs*, 2019. arXiv: 1904.01681 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1904.01681.

[105] A. Ghosh, H. S. Behl, E. Dupont, P. H. S. Torr, and V. Namboodiri, *STEER: Simple temporal regularization for neural ODEs*, 2020. arXiv: 2006.10711 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2006.10711.

[106] S. Greydanus, M. Dzamba, and J. Yosinski, *Hamiltonian neural networks*, 2019. arXiv: 1906.01563 [cs.NE]. [Online]. Available: https://arxiv.org/abs/1906.01563.

[107] Q. Kang, Y. Song, Q. Ding, and W. P. Tay, "Stable neural ode with lyapunov-stable equilibrium points for defending against adversarial attacks," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, Curran Associates, Inc., 2021, pp. 14 925–14 937. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2021/file/7d5430cf85f78c4b7aa09813b14bce0d-Paper.pdf.

[108] A. J. Linot, J. W. Burby, Q. Tang, P. Balaprakash, M. D. Graham, and R. Maulik, "Stabilized neural ordinary differential equations for long-time forecasting of dynamical systems," *Journal of Computational Physics*, vol. 474, p. 111 838, 2023, ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2022.111838. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999122009019.

[109] A. Gholami, K. Keutzer, and G. Biros, *ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs*, 2019. arXiv: 1902.10298 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1902.10298.

[110] A. Tuor, J. Drgona, and D. Vrabie, *Constrained neural ordinary differential equations with stability guarantees*, 2020. arXiv: 2004.10883 [eess.SY]. [Online]. Available: https://arxiv.org/abs/2004.10883.

[111] J. Z. Kolter and G. Manek, "Learning stable deep dynamics models," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/0a4bbceda17a6253386bc9eb45240e25-Paper.pdf.

[112] I. D. J. Rodriguez, A. Ames, and Y. Yue, "Lyanet: A lyapunov framework for training neural ODEs," in *International conference on machine learning*, PMLR, 2022, pp. 18 687–18 703.

[113] D. Floryan, *On instabilities in neural network-based physics simulators*, 2024. arXiv: 2406.13101 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2406.13101.

[114] T. Thummerer and L. Mikelsons, "Eigen-informed neural ordinary differential equations: Incorporating knowledge of system properties," *Neurocomputing*, vol. 654, p. 131 358, 2025, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2025.131358. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231225020302.

[115] P. Hartman, "A lemma in the theory of structural stability of differential equations," *Proceedings of the American Mathematical Society*, vol. 11, no. 4, pp. 610–620, 1960.

[116] D. M. Grobman, "Homeomorphism of systems of differential equations," *Doklady Akademii Nauk SSSR*, vol. 128, no. 5, pp. 880–881, 1959.

[117] D. Arrowsmith and C. M. Place, *Dynamical systems: differential equations, maps, and chaotic behaviour*. CRC Press, 1992, vol. 5.

[118] J. G. F. Francis, "The QR transformation a unitary analogue to the LR transformation — part 1," *The Computer Journal*, vol. 4, no. 3, pp. 265–271, Jan. 1961, ISSN: 0010-4620. DOI: 10.1093/comjnl/4.3.265. eprint: https://academic.oup.com/comjnl/article-pdf/4/3/265/1080833/040265.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/4.3.265.

[119] J. G. F. Francis, "The QR transformation — part 2," *The Computer Journal*, vol. 4, no. 4, pp. 332–345, Jan. 1962, ISSN: 0010-4620. DOI: 10.1093/comjnl/4.4.332. eprint: https://academic.oup.com/comjnl/article-pdf/4/4/332/8201663/040332.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/4.4.332.

[120] V. Kublanovskaya, "On some algorithms for the solution of the complete eigenvalue problem," *USSR Computational Mathematics and Mathematical Physics*, vol. 1, no. 3, pp. 637–657, 1962, ISSN: 0041-5553. DOI: https://doi.org/10.1016/0041-5553(63)90168-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/004155536390168X.

[121] T. Thummerer and L. Mikelsons, *Eigen-informed NeuralODEs: Dealing with stability and convergence issues of NeuralODEs*, 2023. arXiv: 2302.10892 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2302.10892.

[122] M. B. Giles, "An extended collection of matrix derivative results for forward and reverse mode algorithmic differentiation," 2008. [Online]. Available: https://people.maths.ox.ac.uk/gilesm/files/NA-08-01.pdf.

[123] G. Wanner and E. Hairer, *Solving ordinary differential equations II*. Springer Berlin Heidelberg New York, 1996, vol. 375.

[124] L. F. Shampine and C. W. Gear, "A user's view of solving stiff ordinary differential equations," *SIAM review*, vol. 21, no. 1, pp. 1–17, 1979.

[125] D. Halliday, R. Resnick, and J. Walker, *Fundamentals of physics*. John Wiley & Sons, 2013.

[126] T. Thummerer, H. Olsson, C. Song, J. Gundermann, T. Blochwitz, and L. Mikelsons, "LS-SA: developing an FMI layered standard for holistic & efficient sensitivity analysis of FMUs," in *Modelica Conferences*, 2025, pp. 681–691.

[127] C. Rackauckas *et al.*, "Composing modeling and simulation with machine learning in julia," in *2022 Annual Modeling and Simulation Conference (ANNSIM)*, 2022, pp. 1–17. DOI: 10.23919/ANNSIM55834.2022.9859453.

[128] A. Abdelrehim, D. Gandhi, S. Yalburgi, A. Bharambe, R. Anantharaman, and C. V. Rackauckas, "Active learning enhanced surrogate modeling of jet engines in JuliaSim," in *AIAA SCITECH 2025 Forum*. 2025. DOI: 10.2514/6.2025-2323. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2025-2323. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2025-2323.

[129] T. Thummerer, L. Mikelsons, and J. Kircher, "NeuralFMU: Towards structural integration of FMUs into neural networks," in *Proceedings of 14th Modelica Conference 2021, Linköping, Sweden, September 20-24, 2021*, M. Sjölund, L. Buffoni, A. Pop, and L. Ochel, Eds., 2021, ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181297.

[130] C. Bertsch *et al.*, "Beyond FMI - towards new applications with layered standards," in *Modelica Conferences*, 2023, pp. 381–388.

[131] T. Thummerer, L. Mikelsons, and J. Kircher. "FMI.jl (github-repository)," Accessed: Feb. 7, 2025. [Online]. Available: https://github.com/ThummeTo/FMI.jl.

[132] T. Thummerer and L. Mikelsons. "FMIFlux.jl (github-repository)," Accessed: Feb. 7, 2025. [Online]. Available: https://github.com/ThummeTo/FMIFlux.jl.

[133] C. Tsitouras, "Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption," *Computers & Mathematics with Applications*, vol. 62, no. 2, pp. 770–775, 2011. DOI: 10.1016/j.camwa.2011.06.002.

[134] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, IEEE, 2009, pp. 248–255.

[135] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[136] M. Towers *et al.*, *Gymnasium: A standard interface for reinforcement learning environments*, 2024. arXiv: 2407.17032 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2407.17032.

[137] J. Thiyagalingam, M. Shankar, G. Fox, and T. Hey, "Scientific machine learning benchmarks," *Nature Reviews Physics*, vol. 4, no. 6, pp. 413–420, 2022.

[138] M. Takamoto *et al.*, "PDEBench: An extensive benchmark for scientific machine learning," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 1596–1611. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/0a9747136d411fb83f0cf81820d44afb-Paper-Datasets_and_Benchmarks.pdf.

[139] P. Ren, N. B. Erichson, S. Subramanian, O. San, Z. Lukic, and M. W. Mahoney, "SuperBench: A super-resolution benchmark dataset for scientific machine learning," *ArXiv*, vol. abs/2306.14070, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259251642.

[140] S. Farrell *et al.*, "MLPerf™ HPC: A holistic benchmark suite for scientific machine learning on HPC systems," in *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, 2021, pp. 33–45. DOI: 10.1109/MLHPC54614.2021.00009.

[141] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018.

[142] T. Thummerer and L. Mikelsons. "Scientific machine learning using functional mock-up units," Accessed: Feb. 7, 2025. [Online]. Available: https://pretalx.com/juliacon2024/talk/9KQJFS/.

[143] T. Thummerer and L. Mikelsons. "Hybrid modeling using FMI," Accessed: Feb. 7, 2025. [Online]. Available: https://cps-vo.org/node/96930.

[144] P. Crosthwaite, "Clockwork automata, artificial intelligence and why the body of the author matters," in *Minds, Bodies, Machines, 1770–1930*, Springer, 2011, pp. 83–104.

[145] Andrés F. Ruiz H. "Servomechanisms." Modelica library, Accessed: Oct. 17, 2025. [Online]. Available: https://github.com/afrhu/Servomechanisms.

[146] D. Zimmer, F. van der Linden, Z. Qu, and German Aerospace Center (DLR), Institute of System Dynamics and Control (SR), *Modelica library PlanarMechanics*, https://github.com/dzimmer/PlanarMechanics/releases/tag/v1.6.0, version 1.6.0, Version 1.6.0, Sep. 2023.

[147] T. Thummerer, F. Jarmolowitz, D. Sommer, and L. Mikelsons, "Br(e)aking the boundaries of physical simulation models: Neural functional mock-up units for modeling the automotive braking system," in *Modelica Conferences*, 2025, pp. 575–583.

[148] R. Bellman and R. Kalaba, "On adaptive control processes," *IRE Transactions on Automatic Control*, vol. 4, no. 2, pp. 1–9, 1959. DOI: 10.1109/TAC.1959.1104847.

[149] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, 1978. DOI: 10.1109/TASSP.1978.1163055.

[150] C. Myers, L. Rabiner, and A. Rosenberg, "Performance tradeoffs in dynamic time warping algorithms for isolated word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 6, pp. 623–635, 1980. DOI: 10.1109/TASSP.1980.1163491.

[151] K. Heun *et al.*, "Neue Methoden zur approximativen Integration der Differentialgleichungen einer unabhängigen Veränderlichen," *Z. Math. Phys*, vol. 45, pp. 23–38, 1900.

[152] A. Rahman, J. Drgoňa, A. Tuor, and J. Strube, "Neural ordinary differential equations for nonlinear system identification," in *2022 American control conference (ACC)*, IEEE, 2022, pp. 3979–3984.

[153] C. Audebert, P. Bucur, M. Bekheit, E. Vibert, I. E. Vignon-Clementel, and J.-F. Gerbeau, "Kinetic scheme for arterial and venous blood flow, and application to partial hepatectomy modeling," *Computer Methods in Applied Mechanics and Engineering*, vol. 314, pp. 102–125, 2017.

[154] P. H. Charlton, J. Mariscal Harana, S. Vennin, Y. Li, P. Chowienczyk, and J. Alastruey, "Modeling arterial pulse waves in healthy aging: A database for in silico evaluation of hemodynamics and pulse wave indexes," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 317, no. 5, H1062–H1085, 2019, PMID: 31442381. DOI: 10.1152/ajpheart.00218.2019. eprint: https://doi.org/10.1152/ajpheart.00218.2019. [Online]. Available: https://doi.org/10.1152/ajpheart.00218.2019.

[155] R. B. Armstrong, M. D. Delp, E. F. Goljan, and M. H. Laughlin, "Distribution of blood flow in muscles of miniature swine during exercise," *Journal of Applied Physiology*, vol. 62, no. 3, pp. 1285–1298, 1987, PMID: 3106313. DOI: 10.1152/jappl.1987.62.3.1285. eprint: https://doi.org/10.1152/jappl.1987.62.3.1285. [Online]. Available: https://doi.org/10.1152/jappl.1987.62.3.1285.

[156] T. Thummerer and L. Mikelsons, "Boosting (existing) simulations with hybrid modeling: Best practices for physics-enhanced neural ordinary differential equations (PeNODEs)," Workshop (unpublished), 2022.

[157] M. Seeger, "Gaussian processes for machine learning," *International journal of neural systems*, vol. 14, no. 02, pp. 69–106, 2004.

[158] Z. Li, J. Ji, and Y. Zhang, "From kepler to newton: Explainable AI for science discovery," *CoRR*, vol. abs/2111.12210, 2021. arXiv: 2111.12210. [Online]. Available: https://arxiv.org/abs/2111.12210.

[159] S. Kim *et al.*, "Integration of neural network-based symbolic regression in deep learning for scientific discovery," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 9, pp. 4166–4177, 2020.

[160] B. K. Petersen, M. Landajuela, T. N. Mundhenk, C. P. Santiago, S. K. Kim, and J. T. Kim, "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients," *arXiv preprint arXiv:1912.04871*, 2019.

[161] M. Heider, H. Stegherr, R. Sraj, D. Pätzel, J. Wurth, and J. Hähner, "SupRB in the context of rule-based machine learning methods: A comparative study," *Applied Soft Computing*, vol. 147, p. 110 706, 2023, ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2023.110706. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S156849462300724X.

[162] L. Ardizzone *et al.*, "Analyzing inverse problems with invertible neural networks," *arXiv preprint arXiv:1808.04730*, 2018.

[163] "Homepage of SSP standard," Accessed: Jan. 2, 2024. [Online]. Available: https://ssp-standard.org/.