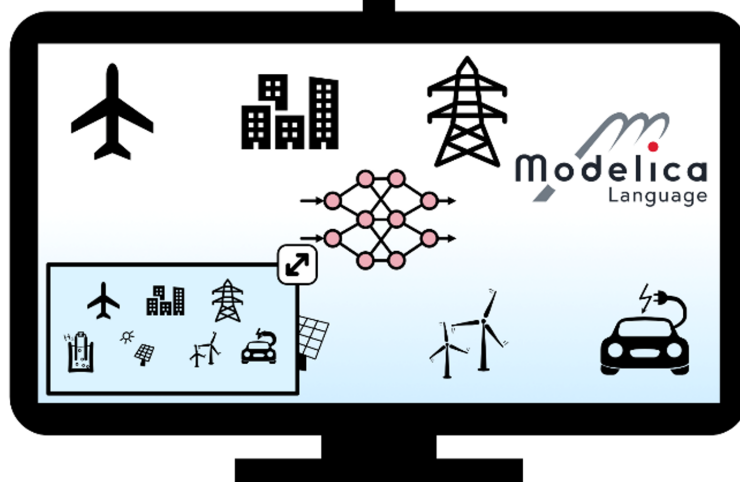


ITEA4 22013



Open standards for  
Scalable Virtual Engineering  
and Operation



# Deliverable D4.1

## *Standard enhancements and tool features for PeN-ODEs*

Julia Gundermann, Tobias Thummerer

Version v1.01, Aug 7th, 2025

# Table of Contents

1. Introduction .....	3
2. Standard Enhancements .....	3
3. Tool Features .....	4
3.1. OpenModelica .....	4
3.2. Dymola .....	4
3.3. Software Production Engineering (former name CATIA ESP) .....	5
3.4. FMIFlux.jl .....	5
3.5. Modia .....	5
3.6. SimulationX .....	5
3.7. TLK tools and libraries .....	6
3.8. OPTIMAX ABB .....	6
3.9. XRG tools and libraries .....	6
Attachments .....	6

# Project Acronyms

Acronym	Description
AI	Artificial Intelligence
COP	Coefficient of Performance
CPS	Cyber Physical System
CSP	Credible Simulation Process
DAE	Differential-algebraic equation
eFMI	Functional Mock-up Interface for embedded systems
FMI	Functional Mock-up Interface (standard for model exchange and co-simulation)
FMI component	A model in FMI format (= FMU)
FMU	Functional Mock-up Unit (= an FMI component)
HVAC	Heating, Ventilation and Air Conditioning
LOTAR	Long Term Archiving and Retrieval
LSS	Large-scale System
M&S	Modelling and Simulation
Modelica	Standard for modelling of cyber-physical systems
MosSEC	Modelling and simulation information in a collaborative system engineering context
MiL/SiL/HiL	Model/Software/Hardware in the Loop
NeuralODE	Ordinary Differential Equation, where a Neural Network (NN) defines its derivative function
NN	Neural Network
ODE	Ordinary Differential Equation
PeN-ODE	Physics enhanced Neural ODE
ONNX	Open Neural Network Exchange standard
PINN	Physics-informed Neural Network
SSP	System Structure and Parameterization (standard for connected FMUs)
UQ	Uncertainty Quantification

# From Full Project Proposal

ID	Type	Description	Due Month	Access
D4.1	Doc	Standard enhancements and tool features for PeN-ODEs	M18	Public



# 1. Introduction

The use of machine learning in modeling physical systems enables more efficient modeling with respect to accuracy, simulation speed, and effort. OpenSCALING focuses on PeN-ODEs (Physically enhanced Neural ODEs), a non-standard application of neural networks that combine NNs and physical equations in ODEs or DAEs. Currently, the widespread use of PeN-ODEs in modeling and simulation of LSS is hindered by the lack of appropriate training strategies, best practices, standards, and tool support. This Deliverable lists standard enhancements and tool features, which will help to move from custom interfaces and toolchains for the training of PeN-ODEs to native support in simulation tools.

## 2. Standard Enhancements

Based on the input requirements from the demonstrators, and the capacities of the contributing parties, the planned standard enhancements focus on extensions of the FMI standard to enable holistic sensitivity analysis. This is essential for efficient and robust training of setups with neural networks and FMUs. Additionally further applications as advanced optimal control will profit from the standard extensions. The extensions are outlined in the paper "LS-SA: Developing an FMI layered standard for holistic & efficient sensitivity analysis over FMUs", submitted by UNA, DS, ABB and Keysight to the 16th International Modelica and FMI Conference. A draft of this paper is attached to this document [\[1\]](#).

Regarding eFMI, first prototype support for PeN-ODEs in Dymola showed that no standard extensions are required. This is backed by demonstrator D4.2-01 developed by DLR and Dassault Systèmes, a neural quarter car vehicle model deployed as predictive controller on an ECU. Here, the physics are modeled in Modelica, afterwards the PeN-ODE is trained offline in PyTorch, and finally reimported as pure Modelica code with the tensor flow modeled as multi-dimensional Modelica expressions in the NeuralNetwork Modelica library. Based on this pure Modelica representation, eFMI tooling (Dymola and Software Production Engineering of Dassault Systèmes) can be used to generate embedded domain suited code without the need for any eFMI standard extensions. To ease the definition of online neural network parameter recalibrations (weights and biases), Dymola has been extended with a new Modelica annotation enabling to mark *nested* parameter records for exposure as eFMI tunable parameters. The approach of D4.2-01 is explained in detail in a paper, which will be published at the 16th International Modelica and FMI Conference [\[2\]](#), and is attached here as a draft.

Beyond FMI and eFMI, extensions in Modelica were discussed, leading to the conclusion that the language holds any required feature to define different layers of neural networks

(as shown for example in [2]), and no further standardization is needed. Proposals to facilitate the definition or training of PeN-ODEs, will not be covered in the project, due to limited resources and test options within the project, including:

- adjoint derivative annotation in Modelica
- function placeholder in Modelica, FMI, eFMI

## 3. Tool Features

### 3.1. OpenModelica

HSBI and LiU develop the following functionality for OpenModelica:

- Runtime for Dynamic Optimization used for PeN-ODE training [3] with following new features:
  - optimization of generic parameter dependent models
  - seamless incorporation of physics aware constraints
  - native Modelica (via NeuralNetwork library) with full DAE support
  - workflow fully integrated in OpenModelica environment (no need to leave the tool)
- FMI functionality `fmiXGetDirectionalDerivative`, `fmiXGetAdjointDerivative` (required for PeN-ODE training) and others on demand
- Support for unscaled (parameter/variable) arrays and equations (WP3)

### 3.2. Dymola

Dassault Systèmes developed the following extensions for Dymola:

- First *prototype* support — to be further improved — to avoid the scalarization of multi-dimensional expressions representing tensor flows (also related to WP3). This is needed to optimize code size for neural networks when generating embedded code via the eFMI facilities of Dymola, for example in demonstrator D4.2-01 [2].
- Derivatives (delivered as part of Dymola 2025x Refresh 1):
  - First public beta version for efficiently generating adjoint derivatives for FMUs (`fmiXGetAdjointDerivative`). This has both been useful in itself, and as input to layered standard [1]. Improvements are needed, in particular events are not handled.

- Both directional and adjoint derivatives now support dynamic state selection.
- New Modelica annotation to expose parameter record instances *nested* within models as tunable eFMI parameters.

### 3.3. Software Production Engineering (former name CATIA ESP)

No improvements so far, but planned are optimizations of generated production code for better supporting multi-dimensional eFMI GALEC expressions.

### 3.4. FMIFlux.jl

UNA keeps developing the Julia library *FMIFlux.jl* that allows for modeling PeN-ODEs based on FMUs. The following is under development:

- full implementation of the LS-SA prototype [\[1\]](#):
  - get/set discrete state
  - time gradients
  - parameter sensitivities (already implemented)
  - efficient (state) sensitivities over event instances
  - error indicators
  - ...
- various optimizations (e.g. usage of FMI dependency information for efficient Jacobian sampling)

### 3.5. Modia

Modia is no longer under development.

### 3.6. SimulationX

Keysight (formerly ESI) develops prototypes of the named functionality for its system simulation tool SimulationX:

- ONNX Runtime Interface to Modelica to import neural networks to system models
- implementation of functionality required for PeN-ODE training: `fmiXGetDirectionalDerivative`, `fmiXGet/SetFMUState`

- prototype implementation of the features that are planned to be standardized in LS-SA:
  - get/set of a discrete state in FMI
  - time gradient
  - parameter sensitivities

## 3.7. TLK tools and libraries

TLK enhances its software products, in particular the TIL Suite, to utilize the standard enhancements and newly developed Modelica tool features. This will enable using the PeN-ODE methodology with system models based on the TIL Suite (e.g. demonstrator 1.4 and 1.6).

## 3.8. OPTIMAX ABB

OPTIMAX is developing PeN-ODEs functionality into the current AutoML (Automatic Machine Learning) module, in order to support the physics-enhanced and data-driven component creation.

- Python-based PeN-ODEs model training workflow / pipeline (PyTorch, FMPy, etc.).
- Creating native modelica model with the help of NeuralNetwork library (HSBI and LiU) within OPTIMAX tooling environment.

## 3.9. XRG tools and libraries

XRG integrates hybrid model generation + PeN-ODE training in its XRG JSCORE application. In particular JSCORE will be extended by automated training + inference of PeN-ODEs as well by a more user friendly geometry and results visualization. Furthermore we will enhance our Modelica libraries SMArtInt / SMArtInt+ to support the newly developed Modelica Standard features and to support computational efficient simultaneous evaluation of large numbers of PeN-ODEs.

# Attachments

Remark: the referenced publications are attached as draft versions to this document. The final and official versions will be available in the Linköping University Electronic Press proceedings of the 16th Modelica Conference.

- [1] "LS-SA: Developing an FMI layered standard for holistic & efficient sensitivity analysis over FMUs" Tobias Thummerer, Hans Olsson, Chen Song, Julia Gundermann,

Torsten Blochwitz, Lars Mikelsons. Submitted to: 16th International Modelica & FMI Conference, Lucerne, Switzerland, Sep 8-10, 2025.

- [2] "Hybrid Simulation Models for Embedded Applications: A Modelica and eFMI approach" Tobias Kamp, Christoff Bürger, Johannes Rein, Jonathan Brembeck  
Submitted to: 16th International Modelica & FMI Conference, Lucerne, Switzerland, Sep 8-10, 2025.
- [3] "Efficient Training of Physics-enhanced Neural ODEs via Direct Collocation and Nonlinear Programming" Linus Langenkamp, Philip Hannebohm, Bernhard Bachmann. Submitted to: 16th International Modelica & FMI Conference, Lucerne, Switzerland, Sep 8-10, 2025.

# LS-SA: Developing an FMI layered standard for holistic & efficient sensitivity analysis of FMUs

Tobias Thummerer<sup>1</sup> Hans Olsson<sup>2</sup> Chen Song<sup>3</sup> Julia Gundermann<sup>4</sup> Torsten Blochwitz<sup>4</sup> Lars Mikelsons<sup>1</sup>

<sup>1</sup>Chair of Mechatronics, University of Augsburg, Germany, {tobias.thummerer,lars.mikelsons}@uni-a.de

<sup>2</sup>Dassault Systemes AB, Sweden, {hans.olsson}@3ds.com

<sup>3</sup>ABB Corporate Research Center, Germany, {chen.song}@de.abb.com

<sup>4</sup>Keysight Technologies Deutschland GmbH, Germany,  
{julia.gundermann,torsten.blochwitz}@keysight.com

## Abstract

The Functional Mock-up Interface (FMI) is *the* standard for exchanging industrial simulation models in a variety of different applications. Although sensitivity analysis for continuously differentiable systems is directly supported by the standard, for systems with state discontinuities, it is only possible to determine correct sensitivities to a limited extent. In this position paper, we investigate how sensitivity analysis for discontinuous Functional Mock-up Units (FMUs), i.e. including state and time events, works in theory and which additional steps are required to obtain correct results in practice. We further investigate that these steps are unnecessarily computationally intensive from a mathematical point of view, but cannot be implemented in a more efficient way under the current restrictions of the standard. We therefore make a concrete proposal for the new *layered standard sensitivity analysis* (LS-SA) that remedies the current deficits of FMI in the sensitivity analysis of discontinuous systems. In this way, LS-SA opens FMI towards a variety of next-level applications — including (scientific) machine learning and optimal control — by providing fully differentiable FMUs under high computational performance.

**Keywords:** *Functional Mock-up Unit, Functional Mock-up Interface, Discontinuous, Events, Machine Learning, Scientific Machine Learning, Neural FMU, Layered Standard*

## 1 Introduction

We start by motivating the actual topic, followed by a brief introduction to the relevant basics to understand the paper.

### 1.1 Motivation & Goal

The FMI standard was created to allow the exchange of standardized models within and outside of company boundaries. The topic of Sensitivity Analysis (SA) has already played a role in the development of the standard, for example, to be able to perform uncertainty assessments on models. The groundbreaking developments in the field of machine learning in recent years have led to an increas-

ing fusion of classical engineering disciplines such as scientific computing with machine learning, establishing the research field of *Scientific Machine Learning* (Baker et al. 2019; Rackauckas, Ma, et al. 2021). The backbone of (scientific) machine learning is efficient SA. Therefore, in our view, enhancing SA capabilities for FMUs is a crucial step toward advancing the standard for future applications in research and industry. This improvement will facilitate the use of FMI in machine learning, advanced dynamic optimization, and other fields.

Today, many industrial optimization platforms are used to improve energy efficiency, reduce operational costs, and improve sustainability in various sectors, for example, ABB OPTIMAX<sup>®</sup> (Franke et al. 2008) and TLK-Thermo (Petr, Tegethoff, and Köhler 2017). Such platforms often operate based on FMUs because of the seamless integration of physics-based models and efficient handling of complex multi-domain systems. Enabling proper SA in FMUs becomes more and more important, because many industrial optimization problems are highly nonlinear, nonconvex and discontinuous.

While FMI already offers possibilities for SA, for example for integration of FMUs into frameworks for Automatic Differentiation (AD) (s. Sec. 3.4), these are only sufficient to perform SA correctly for continuously differentiable systems. However, for discontinuous systems, further measures are necessary for a correct SA, as we will investigate further. This is of crucial relevance for industrial applications, where most models include discontinuities.

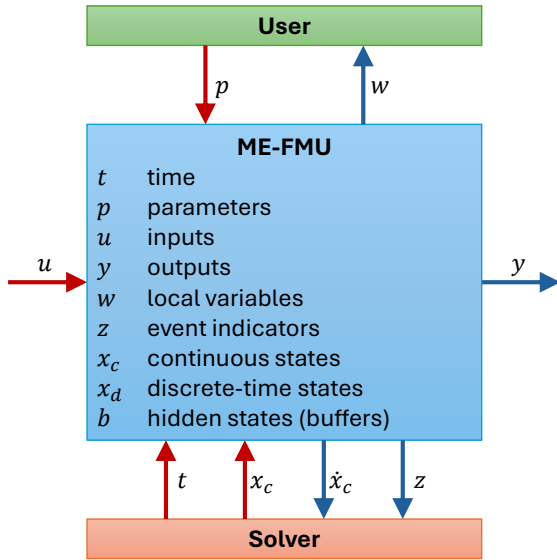
In this position paper, we want to show what is currently possible in FMI and what should be made possible to be open to current and future research fields, e.g., neural FMUs (Thummerer, Mikelsons, and Kircher 2021), the combination of FMUs and Artificial Neural Networks (ANNs). We start by investigating two applications of current research that we believe should be made solvable with FMI. We then examine the current interface offered by FMI and collect requirements with respect to unsolved tasks within these applications. Based on these requirements, we make concrete proposals for an extension of

FMI on the basis of a Layered Standard (LS), i.e. without adjustments to the actual standard release. In parallel, we are already working on a prototype implementation, i.e. the suggestions made are already being tested.

In the following, we provide a brief introduction to the foundational concepts necessary to understand the paper.

## 1.2 Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) is an open standard that defines an interface to exchange dynamic models in the form of a ZIP container which stores the model as a combination of XML files, binaries, and C code. There are three major releases, the most recent version is 3.0.2 (Modelica Association 2024b), while version 2.0.5 might still be the release with the broadest implementation (Modelica Association 2024a). Motivated by different application scenarios for simulation models, FMI supports three different simulation types, while we focus only on the most relevant for SA, which is Model-Exchange (ME) and is further introduced in the next paragraph. Because ME requires external implementation of the solver and event handling routines, incorporation of SA is more straightforward, for example compared to Co-Simulation (CS). Throughout the paper, for readability reasons, we only use the FMI3 command terminology, and are also focusing specifically on the development of the LS for FMI3. However, the proposed adjustments should also be compatible with FMI2 and we try to highlight the necessary adjustments if not.



**Figure 1.** The ME-FMU and its interface signals. Figure adapted from the official standard document (Modelica Association 2024b).

ME-FMUs provide an interface heavily motivated by Ordinary Differential Equations (ODEs) (as further introduced in 1.4) and are therefore paired with ODE solvers to obtain a solution (referred to as *simulation*), see Fig. 1.

The FMI standard facilitates the transfer and exchange of models in simulation tools. However, it has limitations, when the calculation of a model’s sensitivity is required.

**Layered Standard (LS)** The development of the FMI standard is subject to long release cycles to ensure quality and safety of investments, as the standard is widely used in industry. On the other hand, to allow innovation, in FMI 3.0 the concept of Layered Standard (LS) was put in place to be able to introduce new features in an optional and backward compatible way (Bertsch et al. 2023). A LS can add standardized annotations, standardized extra files, and/or new C-API functions.

## 1.3 Sensitivity Analysis (SA)

Sensitivity Analysis (SA) in the context of FMUs involves studying the influence on the quantities computed by the model (e.g., outputs and state derivatives) w.r.t. variations in quantities that are input to the model (like parameters, states, and inputs). Among others, SA is especially useful for parameter estimation, uncertainty quantification, and optimization.

For example, in optimal chemical production scheduling problems, a mixed-integer linear programming (MILP) solver may be often trapped in suboptimal solutions due to strict reactor start-up time constraints. One can use SA to prioritize the most influential constraints to prevent suboptimal schedules. Moreover, SA can support uncertainty quantification and risk assessment, as industrial systems operate under various uncertainties. By evaluating the range of possible outcomes w.r.t. input variations, SA enhances confidence in optimization results. A further practical example is in hydrogen plant sizing, where SA can assess the impact of load fluctuations, future demand uncertainties, and thermal constraints on transformer efficiency and longevity. This reduces the risk of over- or under-sizing, ensuring that transformers operate within safe margins while optimizing cost and reliability (Gutermuth, Primas, and Bitta 2023).

Last but not least, efficient computation of SA is essential for many modeling and analysis tasks, including hybrid modeling (combining physical and machine learning models) and scientific machine learning. Supporting SA for FMUs is therefore a key enabler for integrating large-scale industrial simulation models into data-driven workflows, as for example required in *neural FMUs* (Thummerer, Mikelsons, and Kircher 2021).

## 1.4 Ordinary Differential Equations (ODEs)

ME-FMUs without events can be interpreted straightforward as ODEs. The ODE within this work is denoted with

$$\dot{x} = f(x, p, t), \quad (1)$$

where the function  $f$  is referred to as *right-hand side*,  $x$  the system state,  $\dot{x}$  the system state derivative,  $p$  parameters and  $t$  the time. Note that external inputs to the system could be implemented by augmenting  $x$  or  $p$ . In this work,

we use this formalism for a simplified description of ME-FMUs without events.

To solve an ODE, a variety of numerical integration schemes exist, that are often incorporated into an abstract definition for arbitrary ODE solvers:

$$x(t) = \text{ODESolve}(f, x_0, p, t_0, t), \quad (2)$$

where  $x_0$  and  $t_0$  are the initial state and time to solve the ODE, (Hairer, Nørsett, and Wanner 1992). To obtain a time-discretized solution for  $f$ , *ODESolve* can be called recurrently, leading to a *simulation loop* as in Alg. 1.

---

**Algorithm 1** *ODESim*: Simple simulation loop without events.

---

**Input:** right-hand side  $f$   
**Input:** time points  $t_i$  for  $i \in \{0, \dots, N\}$   
**Input:** initial state  $x_0$ , parameters  $p$   
 $n = 0$  ▷ initialize step counter  
**while**  $t_n < t_N$  **do** ▷ final time not reached  
     $x_{n+1} = \text{ODESolve}(f, x_n, p, t_n, t_{n+1})$   
     $n = n + 1$   
**end while**  
**Output:** states  $X = \{x_0, \dots, x_N\}$

---

In addition, this concept can be extended to ODEs including *events*. Events are points in time where the system is allowed to change discontinuously. The time point of an event can be known in advance or depend on the system state by defining an event indicator (Modelica Association 2024b). However, this requires a different interface for *ODESolve*, because the termination time for the solver is now not known beforehand (Hairer, Nørsett, and Wanner 1992) and (Chen, Amos, and Nickel 2020):

$$t^*, x(t^*) = \text{ODESolveEvent}(f, c, x_0, p, t_0, t). \quad (3)$$

Here,  $t^*$  is the event time and the event function  $c$  is defined as zero at any event time:

$$c(x(t^*), p, t^*) = 0. \quad (4)$$

This is a simplified view, and in practice a common approach is to monitor changes in signs of a vector containing multiple event indicators, instead of an event function stopping exactly at zero. An event often corresponds to a discontinuous change in the function  $f$ . In essence, processing events (so-called *event handling*) captures the following steps: The event time point is determined (by investigation of the event function zero crossing), the solver is stopped, the discontinuous state change is performed (here, by calling the event affect function  $a$ ) and a fresh integration is started from the new state after the event. The event affect function  $a$  computes a new state for after the event  $x(t^+)$  based on the quantities at time  $t^-$  before the event, and is defined as:

$$x(t^+) = a(x(t^-), p, t^-) = 0. \quad (5)$$

In the actual implementation, it is distinguished between events that solely depend on time (*time events*) or also on states (*state events*). A more detailed introduction on the topic of event handling is not relevant for this work and is omitted for brevity.

---

**Algorithm 2** *ODESimEvent* : Simple simulation loop with events.

---

**Input:** right-hand side  $f$   
**Input:** condition  $c$ , affect  $a$   
**Input:** time points  $t_i$  for  $i \in \{0, \dots, N\}$   
**Input:** initial state  $x_0$ , parameters  $p$   
 $n = 0$  ▷ initialize step counter  
 $\tilde{t} = t_0$  ▷ init. current time  
 $\tilde{x} = x_0$  ▷ init. current state  
**while**  $t_n < t_N$  **do** ▷ final time not reached  
     $\tilde{t}, \tilde{x} = \text{ODESolveEvent}(f, c, \tilde{x}, p, \tilde{t}, t_{n+1})$   
    **while**  $\tilde{t} \neq t_{n+1}$  **do** ▷ next sol. time point not reached  
         $\tilde{x} = a(\tilde{x}, p, \tilde{t})$  ▷ handle event  
         $\tilde{t}, \tilde{x} = \text{ODESolveEvent}(f, c, \tilde{x}, p, \tilde{t}, t_{n+1})$   
    **end while**  
     $x_{n+1} = \tilde{x}$   
     $n = n + 1$   
**end while**  
**Output:** states  $X = \{x_0, \dots, x_N\}$

---

Further, in the FMI specification, the concept of *superdense* time is applied, allowing events to be ordered even if they are triggered for the same  $t$ . For reasons of clarity, this was simplified within this work. In general, the FMI specification (Modelica Association 2024a; Modelica Association 2024b) provides a valuable source for this topic in detail.

## 2 Related Work

We now turn to related work on SA for FMUs, with a particular focus on discontinuous systems.

**FMISensitivity.jl** The Julia library *FMISensitivity.jl*<sup>1</sup> implements fully differentiable FMUs using built-in FMI functions as far as available, while sensitivities not considered in FMI are approximated by finite differences. Further, SA relevant to practice, in terms of reasonable computing effort, requires additional but optional FMI features, like `fmi3Get/SetFMUState` for getting and setting the FMU memory state. The implemented workarounds are investigated in more detail in Sec. 3. Before porting the functionality to *FMISensitivity.jl* in 2023, these features were implemented as part of *FMIFlux.jl*.

**Time-freezing** This approach reformulates non-smooth systems with state jumps into systems that appear smooth from the perspective of a numerical solver (Nurkanović, Sartor, et al. 2021). It introduces a pseudo-time variable and clock state to enforce the discontinuities to lay

---

<sup>1</sup><https://github.com/ThummeTo/FMISensitivity.jl>.



in the derivative rather than in the state itself. During the frozen phases, an auxiliary dynamic system evolves the state smoothly in pseudo-time to mimic the effect of the jump. As a result, the original non-smooth problem is more suitable to conventional ODE solvers without adding integer variables. However, constructing such an auxiliary differential equation is not trivial, and introducing pseudo-time variables may increase the computational effort.

**Finite Elements with Switch Detection (FESD)** This discretization scheme embeds switch detection directly into the time discretization process (Nurkanović, Sperl, et al. 2024). Essentially, it detects state jumps using complementarity constraints and the collocation method. Hence, FESD is able to pose the time points exactly where events happen without relying on the event handling process. The main challenges remain the implementation overhead of a robust switch detection mechanism within a finite element framework and the computational complexity due to the increase of optimization variables and constraints. An implementation of this, as well as the time-freezing approach, can be found as part of the NOSNOC software package (Nurkanović and Diehl 2022).

**CasADi** In (J. Andersson and Goppert 2024), the integration of SA for discontinuous systems within the *CasADi* tool (J. A. Andersson et al. 2019) is investigated. The paper also gives a rough outlook on how to make discontinuous SA available for FMUs under similar considerations to those we make in the course of this work. However, the authors omitted existing work regarding SA for FMUs: Integrate-then-differentiate, as well as differentiate-then-integrate approach, is available since 2021<sup>2</sup> as part of *FMIFlux.jl* and since 2023 as part of *FMISensitivity.jl*, both providing seamless integration with *SciMLSensitivity.jl* (Rackauckas, Ma, et al. 2021) and *DiffEqCallbacks.jl* (Rackauckas and Nie 2017) that provide implementations for a multitude of SA approaches.

**Finite Differences** The most obvious approach to SA would be to perform numerical differentiation of the output of *ODESolve*. There are two main disadvantages: a high amount of numerical noise which can be avoided by freezing the step-sizes and number of iterations or alternatively by internally differentiating (Hairer, Nørsett, and Wanner 1992), and that the cost increases linearly with the number of parameters — using adjoint equations avoids this second issue. On the other hand, there is the great advantage that even systems with discontinuities can be analyzed, which is why this method can be found in practical application despite its poor computational performance.

However, all the investigated solutions are suboptimal from a mathematical perspective and are either not generally applicable or scale insufficiently with the size

of the problem due to limitations in FMI. This strongly motivates the derivation of an addition to the standard (Layered Standard), that is able to provide access to computationally efficient SA, while still being fully backward compatible with FMI.

### 3 Requirements Analysis

In this section, we want to derive requirements for the LS. The essential part of the requirements is the availability of the various Jacobians that are needed for a correct SA. However, it is important to remember that in many cases we are actually interested in *results* of operations *including* the Jacobian, rather than the Jacobian itself. This especially captures Jacobian-Vector Products (JVPs) and Vector-Jacobian Products (VJPs). Note that JVPs are not only useful for SA, but also for solving large ODEs (Ascher and Petzold 1998) with implicit solvers using iterative approaches (e.g., Krylov subspace methods).

We start collecting requirements by investigating two use cases that should be enabled with the new LS. Both applications share the question posed by SA, but look at the actual problem from two different perspectives: while a *parameter optimization* task (s. Sec. 3.1) is defined by the problem of identifying sensitivities for variables *within* the FMU, the *neural FMU* approach (s. Sec. 3.2) requires determination of sensitivities for variables *outside* the actual FMU. As a result, different sensitivities are required to solve both tasks.

#### 3.1 Parameter Optimization

One of the most common applications of SA in the field is the optimization of ODE parameters within a model, often referred to as *model calibration*. When optimizing ODEs, an objective  $l$  is defined that is either minimized (e.g., *loss* in supervised machine learning) or maximized (e.g., *reward* in reinforcement learning). Typically, this loss is not defined on the right-hand side of the ODE itself, but on the *solution* of the ODE.

The gradient of a loss function defined on the ODE solution  $X$ , can be derived by applying the chain rule:

$$\frac{dl(X(p), p)}{dp} = \frac{\partial l(X(p), p)}{\partial X(p)} \frac{\partial X(p)}{\partial p} + \frac{\partial l(X(p), p)}{\partial p}, \quad (6)$$

where the ODE solution  $X$  can stem from *ODESim*, as well as *ODESimEvent*.

**SA for *ODESim*** As can be investigated in algorithm 1, the consecutive evaluation of *ODESolve* leads to the determination of  $X$ . On derivative level, this results in a chain of Jacobians containing

$$\frac{\partial x_{n+1}}{\partial x_n} = \frac{\partial \text{ODESolve}(f, x_n, p, t_n, t_{n+1})}{\partial x_n}, \quad (7)$$

growing with every step of *ODESolve* that needs to be performed to determine the required Jacobian  $\partial X / \partial p$ . Most

<sup>2</sup>Experimental 2021, stable release in 2022 (Thummerer, Stoljar, and Mikelsons 2022).

ODE solvers, for example explicit and implicit Runge-Kutta methods, determine  $x_{n+1}$  by one or more evaluations of  $f$ . This results in differentiation of  $f$  with respect to state and parameters, so  $\partial f/\partial x_n$  and  $\partial f/\partial p$ . These Jacobians can also be found by investigating the sensitivities required for the continuous adjoint method (Céa 1986; Serban and Hindmarsh 2003; Sapienza et al. 2024). If the right-hand side depends on time, further the time gradient  $\partial f/\partial t$  is required. Of course, to obtain a complete derivative chain, also the arithmetic operations inside the ODE solver need to be differentiated. However, the detailed chain of Jacobians depends on the type and implementation of the numerical solver and is therefore omitted.

**SA for *ODESimEvent*** Very similar to *ODESim*, sensitivities for *ODESimEvent* can be collected by investigation of algorithm 2. However, there are differences: First, *ODESimEvent* also determines the event time  $t^*$  besides the system state, therefore differentiation over *ODESolveEvent* results in determination of the Jacobian  $\partial x^*/\partial x_n$  and the additional time gradient  $\partial t^*/\partial x_n$ . Second, the event time  $t^*$  depends on the zero crossing of the event condition  $c$ , which involves differentiation of the event condition with respect to its arguments, so  $\partial c/\partial x_n$  and  $\partial c/\partial t$ . These sensitivities can also be found by investigating the continuous adjoint method for event-ODEs (Chen, Amos, and Nickel 2020; Pfeiffer 2008). Finally, also the derivative of the new state after the event is needed by differentiation of the *affect* function  $a$ , in particular the Jacobian of the state after the event w.r.t. to the state before the event  $\partial x^+/\partial x^-$ .

**SA for calculated Parameters** In practice there are often also calculated parameters  $b = g(p)$  that are computed at the start of the simulation. These calculations may be trivial (e.g., calculating the mass based on the density and volume parameters), or complicated, as, for example, calculating polynomial coefficients based on curve fitting. The set of parameters can be extended to calculated parameters to accumulate sensitivities with respect to  $b$  and then only compute the VJPs for  $\partial g/\partial p$  once. Differentiating these functions should be straightforward even if they are not needed for analytic Jacobians. Parameter subexpressions can be treated together with calculated parameters, as we are not interested in individual calculated parameters but only in their overall impact.

### 3.2 Neural FMUs

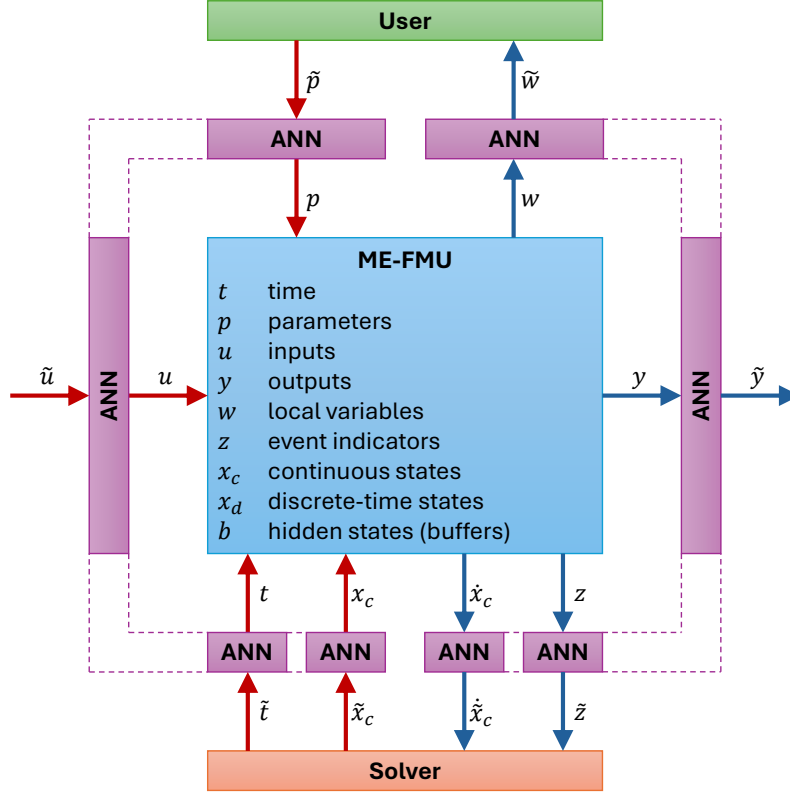
In (Thummerer, Mikelsons, and Kircher 2021), neural FMUs were introduced as the combination of FMUs, ANNs and a numerical ODE solver. In the easiest case, only a single FMU and a single ANN are connected. Even if a common scenario is the modification of the FMU state derivative by the ANN, a variety of different combination schemes are possible. Whereas neural FMUs are in general not restricted to ME-FMUs, we only focus on ME within this work, because we see this as more relevant for the topic of proper SA, and it is definitely more relevant

within the topic of neural FMUs. However, we give a short outlook on CS in the conclusion section.

As introduced, within neural FMUs, no restrictions are made with respect to the connections between the FMU and ANN(s). To make this more tangible, in Fig. 2 all possible positions to extend a ME-FMU by ANNs are given. In addition to the theoretical possibility of doing this, there is of course also the question of practical relevance, which we shall examine below by a brief investigation of fields of applications:

- One of the first applications of hybrid modeling (in terms of combining physical and machine learning models) was the **correction of parameters** by an ANN (Psychogios and Ungar 1992). This is still a meaningful measure within similar applications.
- The most valuable point of intervention is the **correction of state derivatives**. Note that this allows, for example, to incorporate additional forces (like friction) to a system because forces act on the state derivative (acceleration). Manipulation of the state derivative allows us to change the entire behavior of the system because it allows us to overwrite or extend the actual right-hand side of the ME-FMU.
- Time can be manipulated to induce, e.g., **time shifts** to compensate for unsynchronized measurements. In the parallel contribution (Thummerer, Jarmolowitz, et al. 2025), this is investigated.
- By changing the **system input**, corrections can be made to faulty signals. This further opens up the topic of learning an input trajectory to achieve *optimal control*.
- If only the **system outputs** (or local variables) are changed, the correction is learned on top of the simulation performed, without actually influencing the ODE solution, leading in general to less generalization within the ANN. However, this can still be useful in applications where generalizability is less important.
- Manipulation of event indicators allows, for example, for **augmentation of the event indicators** to learn for not modeled event conditions and therefore new discontinuities.
- **Correcting the system state** before passing it to the FMU allows for correcting modeling errors like general shifts in state-space (Thummerer, Mikelsons, and Kircher 2021) or equilibrium position for fluid simulations such as in (Thummerer, Tintenherr, and Mikelsons 2021).

Based on the finding that such architectures have practical relevance, the question arises as to which sensitivities are required for optimization including such architectures.



**Figure 2.** The neural FMU (ME), showing possible intervention points where ANNs can be incorporated to extend the ME-FMU. Individual ANNs could be used for each signal, or all causal input ANNs ( $p$ ,  $u$ ,  $t$  and  $x_c$ ) could be merged to obtain a single input ANN (purple, dashed silhouettes). The same applies for the causal output ANNs ( $w$ ,  $y$ ,  $\dot{x}_c$  and  $z$ ). The discrete-time state  $x_d$  and the buffer  $b$  are not accessible from outside the FMU within the standard definition, granted access to  $x_d$  is part of the LS-SA, however.

Since all possible permutations of existing ANNs are to be investigated, it is expedient to assume the most complex case: All ANNs are incorporated.

### 3.3 Required Sensitivities

Based on the two fields of applications introduced and the corresponding observations, all necessary sensitivities are collected below. It is required to allow for JVP and VJP operations involving the Jacobians<sup>3</sup>  $\partial\gamma/\partial v$ , where  $v \in \{t, x_c, u, p\}$  and  $\gamma \in \{\dot{x}_c, z, y, w\}$ . Furthermore, the actual event affect function that computes the new state of the system after event  $x_c^+$  based on the system state before the event  $x_c^-$  is required but neglected in the picture to be compatible with the illustration in the standard specification. Furthermore, sensitivities are required not only for the continuous state, but also for the entire state of the system, which additionally requires the discrete state  $x_d$ . In summary, the required sensitivities are  $v \in \{t, x_c^-, x_d^-, u, p\}$  and  $\gamma \in \{\dot{x}_c, x_c^+, x_d^+, z, y, w\}$ .

### 3.4 Available methods within FMI

In FMI, there are two commands available that allow to access partial derivatives over an FMU.

<sup>3</sup>For the case of  $\gamma = t$  (scalar time) it is more common to refer to a *gradient* than *Jacobian*.

First, `fmi3GetDirectionalDerivative` allows for JVP-multiplication with a custom seed vector, while `fmi3GetAdjointDerivative` provides functionality for VJP-multiplication. Although this does not allow one to directly access the Jacobians, it allows a straightforward integration within AD frameworks. Forward-mode AD involves consecutive evaluations of JVPs, which allows for a natural propagation of the current sensitivity seed through the FMU with `fmi3GetDirectionalDerivative`. The other way round, for reverse-mode AD, consecutive VJP are computed and `fmi3GetAdjointDerivative` provides an interface for backpropagation of the adjoint sensitivity (or co-tangent) through the FMU. However, there are two issues with these commands: First, the implementation is optional, and, therefore, only a fraction of tools support these commands. Second, and even more important, not all required sensitivities (value references) are accessible by these commands, even if they are implemented. This is investigated in detail, along with solution proposals, in the next section.

## 4 Layered Standard

In the following subsections, we investigate all open issues derived from the introduced applications. For each issue, we will highlight the problem and available workarounds.

Workarounds already implemented within *FMISensitivity.jl* and *FMIFlux.jl* are briefly described below. Finally, possible solutions and their technical implementation in the form of a LS are investigated. We explicitly try to reuse the existing command interface as much as possible, as we want to minimize the implementation effort in the tools and enable a wide adaptation of the LS-SA.

#### 4.1 Event condition

The FMI standard allows for the calculation of derivatives of state derivatives  $\dot{x}$  and outputs  $y$  with respect to states  $x$  and inputs  $u$ . If the system is subject to events, a complete SA requires the sensitivity of the point in time where an event occurs. As indicated in subsection 3.1, this requires derivatives of the event condition, or as it is named in FMI, the event indicator. Since FMI3, event indicators are part of the model variables (so they are exposed), which was not the case for FMI2. Within the FMI specification, there is no restriction that prohibits providing sensitivities for event indicators via the directional or adjoint derivative commands, however it is also not enforced.

**Workaround** The only generally applicable workaround to obtain partial derivatives w.r.t. event indicators is to sample them via finite differences. This way it is implemented in *FMISensitivity.jl*. The required number of samples scales linearly with the Jacobian size to multiply with, that is, linearly with the number of rows for forward AD or the number of columns for reverse AD.

**Proposed Solution (LS-SA)** The proposed solution for the LS-SA is straight-forward: Because the model variables of the event indicators are already known in FMI3, it is only necessary to provide sensitivities for the event indicators via `fmi3GetDirectionalDerivative` and `fmi3GetAdjointDerivative`. For FMI2, it is additionally required to expose the event indicators in the same way as they are exposed in FMI3, by adding the corresponding entries to the section `<ModelStructure>` in the model description.

#### 4.2 Discontinuous State Change

After the event is handled, the numerical integration is restarted with a new state  $x^+$ . However, also this new state may depend on the state before the event  $x^-$ , time, inputs before the event and/or parameters. These sensitivities are not present within FMI — in general, it is not possible to access sensitivities w.r.t. to quantities *before* an event after event handling was performed.

**Workaround** One could sample the required sensitivity by restarting the simulation and disturbing the investigated variable right before the event (finite differences). This is computationally infeasible, because this requires a new simulation for every required sensitive state / input / parameter.

In *FMIFlux.jl*, a much more performant approach is implemented, however, this requires the optional FMI features `fmi3Get/SetFMUState`. If these commands are

available, a memory snapshot of the FMU can be made right before the event. In this way, sampling becomes much more efficient because the snapshot can be used to reinitialize right before the event to disturb the signals for sampling, instead of starting a new simulation. In this way, the complexity reduces to only performing one evaluation of the system of equations rather than performing an entire simulation.

**Proposed Solution (LS-SA)** The challenge is that the  $x^+$  and  $x^-$ , which are required for the Jacobian  $\partial x^+ / \partial x^-$ , represent the same state (model variable) at different locations in (super-dense) time. In FMI, there is no mechanism to access partial derivatives w.r.t to the same variable at different points in super-dense time, because the state before and after the event have the same value reference.

However, in the Modelica modeling language, previous values before event handling can be accessed using the function `pre`. We therefore propose to introduce new model variables that can be used to access previous values of states. This captures the continuous, as well as the discrete-time state. Because derivatives of continuous states are introduced with the `derivative` keyword within the definition of the model variable, we propose a new keyword `previous` for previous states before the event. A code snippet for this is given in Listing 1.

**Listing 1.** Example for a state definition within the model description, including the new *previous* attribute.

```
<Float64
  name="mass.s"
  valueReference="33554432"
  description="absolute position of mass"
  unit="m"
</Float64>
<Float64
  name="der(mass.s)"
  valueReference="587202560"
  description="derivative of mass.s"
  unit="m/s"
  derivative="33554432">
</Float64>
<Float64
  name="pre(mass.s)"
  valueReference="587202561"
  description="previous value for mass.s"
  unit="m"
  previous="33554432">
</Float64>
```

In case of multiple events occurring at the same point in time, the `previous` model variable actually refers to the *previous* value before the last event handled, and not the value before the first event at the time instant. Finally, note that this measure is not necessary for other relevant inputs like  $u$  and  $p$ , because they do not change in super-dense time, for example  $\partial x^+ / \partial p^- = \partial x^+ / \partial p^+$ .

#### 4.3 Time Gradients

As investigated, the gradient with respect to time is needed. Since FMI3, time (or the independent variable

in general) is registered as special model variable with `causality="independent"`. However, there is no enforced possibility within FMI to request partial derivatives w.r.t. time.

**Workaround** As for the event condition, the influence of time, e.g. on the state derivative or outputs, can be sampled straightforwardly during continuous time by disturbing only the time and re-evaluation of the FMU.

**Proposed Solution (LS-SA)** We would like to enforce that directional and adjoint derivatives are allowed to be requested w.r.t. the independent variable during continuous time.

#### 4.4 Event Time

In addition to event indicators that depend on state and time, events can be registered solely depending on time. The time of these events is known exactly, but only step-wise, the next event time is determined during initialization (the first time event) or during event handling (for the upcoming time events). Such events are not determined by zero crossings of event indicators, hence their influence on the FMU's outputs has to be determined separately.

**Workaround** To calculate the influence on the current state or parameter on the next event time, a sampling-based approach is used as for the state change in the previous Sec. 4.2: Before actually handling the current time event, a memory snapshot (`fmi3GetFMUState`) is performed, and the influence of the state and parameters can be successively sampled by disturbing one input to SA, handling the time event and quantifying the influence on the determined event time. This requires at least one sample (one event handling procedure) for every state and parameter in the system and is therefore expensive for large simulations.

**Proposed Solution (LS-SA)** As outlined, the next event time may also depend on the current time. Therefore, it is not sufficient to require the time to be defined as model variable (cf. Sec. 4.3), but necessary to require an additional one for the *next* event time. This enables to calculate the sensitivity of the next event time w.r.t. state and parameters as well as the influence of the current event time on the next event time. Although these sensitivities are required for SA at the next event, they already have to be calculated at the current event and need to be cached.

#### 4.5 Parameter Sensitivities

Even if FMI supports functions to obtain directional derivatives, it is important to state that parameter sensitivities are not available during the actual simulation. Even tunable parameters are only available in event mode (Modelica Association 2024b), which is not useful during continuous-time simulation. This is of course a huge problem and actively prevents parameter SA for FMUs. There are also good reasons to not treat these parameters as tunable, as we are often (e.g., in case of parameter SA)

considering the impact for the entire simulation. Furthermore, tunable parameters can complicate the storage of the memory state, and SA requires storing a large number of memory states.

**Workaround** To our knowledge, there is no obvious workaround to this. However, we observed that some FMU implementations do not implement this limitation and provide correct parameter sensitivities even during continuous-time mode.

**Proposed Solution (LS-SA)** The proposed solution is straightforward: Within the LS, we enforce that parameter sensitivities (at least for tunable parameters) must be accessible via the already existing interface for directional and adjoint derivatives.

Beyond the listed issues, we would also like to add requirements for the application of neural FMUs in the field. In addition to the mathematical considerations for a comprehensive SA, the optimization or training process can be made more efficient and robust with the following enhancements.

#### 4.6 Get/Set Discrete State

A common technique in the training of neural FMUs, and in machine learning in general, is mini-batching. Training data is divided into chunks called *mini-batches*, and parameter gradients are determined for each chunk rather than the entire dataset. A new batch is selected for each training step, either algorithmically or randomly.

If mini-batching is paired with neural FMUs, or mixed continuous-discrete systems in general, a new challenge pops up: The system needs to be initialized for the initial state of every mini-batch. This covers the continuous state, but also the discrete-time state of the FMU. Note that the discrete state can influence the system dynamics, in the extreme case, the discrete state can be used to switch between completely different sets of equations for the continuous right-hand side. In this case, the discrete state is sometimes referred to as *mode* of the system.

By default, FMI completely hides the discrete-time state of the FMU. It is neither readable nor writable from the outside. However, this is required to initialize the system based on data. However, it is important to state that for proper initialization the continuous and discrete states must *match*. Depending on the system model, not every combination of continuous and discrete states is meaningful and leads to a solvable system.

**Workaround** For now, FMI does not provide an interface to directly access the discrete state, but an interface to catch it *indirectly*. The corresponding commands are `fmi3Get/SetFMUState`, which create (or activate) a memory copy of the entire FMU memory state, including the continuous and discrete state, but also additional variables, such as starting values for iteration of algebraic loops. It is also important that commands for the memory

states are declared *optional* in the FMI specification and only a fraction of tools implemented these.

Furthermore, while these commands can be used to *re-visit* a discrete state after capturing it once, they do not allow one to actually modify the discrete state. For example, if the system needs to be initialized in a very specific discrete state, one needs to *guide* the system to this specific state, e.g. by controlling inputs, in order to capture the intended discrete state. This is an immense effort and is generally not feasible in the case of a complex simulation model. This workaround should therefore only be regarded as a makeshift solution.

**Proposed Solution (LS-SA)** Even if this step is associated with technical challenges, we see no alternative to allowing reading / writing of the discrete state of the system, as is allowed for the continuous part of the state within FMI. We propose the following additions to FMI:

1. Because the data type of the discrete state may vary, compared to the continuous state which is always `Float64` / `Float32`, we do not suggest introducing new commands such as `fmi3Get/SetDiscreteState`. We propose to reuse the existing commands for reading/writing variables of the FMU, like e.g. `fmi3Get/SetInt32`. This way, no new commands are introduced, which we see as the main obstacle when implementing a LS, and there is no problem with multiple data types used for the discrete state, as it can be set/get by multiple consecutive calls (e.g. mixing boolean and integer values).
2. The value references of the discrete states must be made available from outside the FMU, in order to allow for setting/getting the discrete state. We propose to add this information to the model description, as part of the section `<ModelStructure>`. There, we like to add a new section `<DiscreteState>`, where value references of discrete states are listed in the same way as, e.g. for outputs. Because FMI does not allow us to add new fields to the standardized model description, the addition of fields does not happen within the existing model description. Technically speaking, a separate file is created consisting only of the new fields and a minimum representation of the model description XML tree around.
3. Discrete states are only required to be settable in event mode (where the discrete state changes during regular simulation), and we propose to require triggering event handling consecutively to ensure that the given continuous states match the discrete states.

## 4.7 Comply with Assertions

The use of assertions when modeling large systems is a meaningful measure in developing correct and easily maintainable simulation models. Typically, assertions are

formulated in binary form, i.e. they remain unnoticed until a condition is violated. The existence of an assertion and the monitored condition cannot be seen outside the FMU. This has drastic effects on the application of hybrid modeling with FMUs (e.g. neural FMUs). If, for example, the system dynamics is changed by an ANN, the violation of a condition within an assertion can only be detected when it is already too late and an exception is thrown, which often leads to the termination of the simulation and thus the SA.

To our knowledge, there is **no workaround** to this.

**Proposed Solution (LS-SA)** We propose to introduce a concept related to the event indicator interface, which we call straightforward *error indicators*. Error indicators provide a distance measure for the violation of modeling errors. Unlike event indicators, the sign is important. Negative error indicators identify the non-existence of a modeling error, whereas positive error indicators show that a modeling error occurred. For the case of at least one positive error indicator, we do not expect the FMU to provide meaningful values, e.g. returning `fmi3StatusError` or `fmi3StatusDiscard` for calls to `fmi3GetX`. To implement error indicators, we propose the following modifications:

1. Error indicators are introduced as model variables in the same way as event indicators, and are made public in the section `<ModelStructure>` in the model description, where we add a new subsection `<ErrorIndicator>`, again in the same way as for `<EventIndicator>`. As for the discrete state in 4.6, this measure is implemented in a parallel model description file and not by modification of the standardized FMI model description.
2. To make the values of error indicators accessible, two obvious strategies are available. First, the error indicators could be accessed through `fmi3GetFloat64` as any other (`Float64`) model variable. For now, we see no reason to not enforce all error indicators to be of type `Float64` (for 64-bit FMUs). Second, because all event indicators share the same data type, a new command could be introduced in analogy to `fmi3GetEventIndicators`, see code 2, that allows capturing all error indicators at once.
3. As proposed for event indicators, we also like to keep error indicators differentiable. Although this is not required for proper SA, this opens up an interesting and valuable new use case: Explicit regularization. If not only the distance to an error is known, but also in which direction parameter changes influence the error (*gradient*), error indicators can be used to actively prevent errors during training, for example by adding the (differentiable) error indicators to the training objective as additional summands (referred to as *regu-*

larization terms). Sensitivities could be accessed in analogy to the event indicators.

**Listing 2.** The new error indicator function.

```
typedef fmi3Status fmi3GetErrorIndicators(  
    fmi3Instance instance,  
    fmi3Float64 errorIndicators[],  
    size_t nErrorIndicators);
```

Although we consider all the proposed measures within this section important, we have sorted them in descending priority order (with the aim of carrying out a correct SA) in order to support a targeted implementation.

## 5 Conclusion

In summary, we have investigated the opportunities that already exist for SA in FMI. By investigating two example applications (parameter SA and neural FMUs), we were able to identify deficits, particularly in the analysis of discontinuous systems. We went on to consider how we could close these gaps (mathematically and technically) and made a concrete proposal for the LS-SA, which will be able to solve the issues investigated under reasonable computational performance.

**Current & Future Work** Even if the LS is not yet fully implemented, a dedicated working group within the research project *OpenSCALING*, including tool vendors, users, and academia, continues to develop and implement the LS to be able to publish it during the project period. As soon as more prototypes of tool implementations are available, we want to examine and quantify the practical computational benefits of the LS besides the theoretical considerations.

One major part of future work is a more detailed investigation and specification of the proposed measures, e.g. it is necessary to specify allowed (or forbidden) attributes for the newly introduced model variables.

Of course, the development of LS-SA is also strongly driven by *specific* applications for which we are aiming, which will benefit greatly from the realization of the standard extension. For example, training of neural FMUs, the combination of ANNs and physics-based simulation models captured in an FMU, will benefit from a significantly faster training process (multiple factors, depending on the number of states and events).

In theory, the proposed LS-SA can be extended for CS-FMUs. However, this introduces additional challenges because parts of SA, which are now part of the simulator, need to be incorporated to the FMU. Further, a clever interface needs to be discussed that allows for efficient determination of sensitivities, for example by defining required sensitivities before starting the actual simulation (e.g. during the setup of experiment).

This publication lays the methodological foundation for further discussions on a large and importing topic within the standard, as well as prototype implementations

within importing and exporting tools. It is important for us to point out that the LS is in active development, but not officially released. We would like to draw the community's attention to the development, as well as the intended measures, and we hope to achieve a broad discussion of this topic in the community so that a strong LS with broad adoption can emerge from this. We welcome any kind of participation.

## Funding

This research was partially funded by ITEA4-Project OpenSCALING (Open standards for SCALable virtual engineeriNG and operation) No. 22013. For more information, see: <https://openscaling.org/> (accessed on 30 April 2025).

## Author Contributions

Conceptualization, T.T., O.H., S.C., G.J., B.T., M.L.; methodology, T.T., O.H., G.J., B.T., M.L., S.C.; writing—original draft preparation, T.T., O.H., S.C., G.J., B.T., M.L.; writing—review and editing, T.T., O.H., S.C., G.J., M.L.; visualization, T.T.; All authors have read and agreed to the published version of the manuscript.

## Abbreviations

**AD** Automatic Differentiation

**ANN** Artificial Neural Network

**CS** Co-Simulation

**FMI** Functional Mock-up Interface

**LS** Layered Standard

**FESD** Finite Elements with Switch Detection

**FMU** Functional Mock-up Unit

**JVP** Jacobian-Vector Product

**ME** Model-Exchange

**MILP** mixed-integer linear programming

**ODE** Ordinary Differential Equation

**SA** Sensitivity Analysis

**VJP** Vector-Jacobian Product

## References

- Andersson, Joel and James Goppert (2024). “Event support for simulation and sensitivity analysis in CasADi for use with Modelica and FMI”. In: *Modelica Conferences*, pp. 99–108.
- Andersson, Joel AE et al. (2019). “CasADi: a software framework for nonlinear optimization and optimal control”. In: *Mathematical Programming Computation* 11, pp. 1–36.



- Ascher, Uri M. and Linda R. Petzold (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Philadelphia, PA: Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611971392.
- Baker, Nathan et al. (2019). *Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence*. Tech. rep. USDOE Office of Science (SC), Washington, DC (United States).
- Bertsch, Christian et al. (2023). “Beyond FMI-Towards New Applications with Layered Standards”. In: *Modelica Conferences*, pp. 381–388.
- Céa, Jean (1986). “Conception optimale ou identification de formes, calcul rapide de la dérivée directionnelle de la fonction coût”. In: *Revue Française d’Automatique, Informatique, et Recherche Opérationnelle (RAIRO)* 20.4, pp. 371–402.
- Chen, Ricky T. Q., Brandon Amos, and Maximilian Nickel (2020). “Learning Neural Event Functions for Ordinary Differential Equations”. In: *CoRR* abs/2011.03902. arXiv: 2011.03902. URL: <https://arxiv.org/abs/2011.03902>.
- Franke, Rüdiger et al. (2008). “Model-based online applications in the ABB Dynamic Optimization framework”. In: *Proc. of the 6th Int. Modelica Conference, Bielefeld*. [www.modelica.org/events/-modelica2008/Proceedings/sessions/session3b1.pdf](http://www.modelica.org/events/-modelica2008/Proceedings/sessions/session3b1.pdf).
- Gutermuth, Georg, Bernhard Primas, and Jan Bitta (2023). “Sizing matters”. In: *ABB Review*. URL: <https://new.abb.com/news/detail/103158/sizing-matters>.
- Hairer, E., S.P. Nørsett, and G. Wanner (1992). *Solving Ordinary Differential Equations I Nonstiff problems*. Second. Berlin: Springer.
- Modelica Association (2024a-11). *Functional Mock-up Interface for Model Exchange and Co-Simulation. Document version: 2.0.5*. Tech. rep. Linköping: Modelica Association. URL: <https://github.com/modelica/fmi-standard/releases/download/v2.0.5/FMI-Specification-2.0.5.pdf>.
- Modelica Association (2024b-11). *Functional Mock-up Interface Specification. Version 3.0.2*. Tech. rep. Modelica Association. URL: <https://fmi-standard.org/docs/3.0.2/>.
- Nurkanović, Armin and Moritz Diehl (2022). “NOSNOC: A Software Package for Numerical Optimal Control of Nonsmooth Systems”. In: *IEEE Control Systems Letters* 6, pp. 3110–3115. DOI: 10.1109/LCSYS.2022.3181800.
- Nurkanović, Armin, Tommaso Sartor, et al. (2021). “A Time-Freezing Approach for Numerical Optimal Control of Nonsmooth Differential Equations With State Jumps”. In: *IEEE Control Systems Letters* 5.2, pp. 439–444. DOI: 10.1109/LCSYS.2020.3003419.
- Nurkanović, Armin, Mario Sperl, et al. (2024). “Finite elements with switch detection for direct optimal control of nonsmooth systems”. In: *Numerische Mathematik* 156.3, pp. 1115–1162.
- Petr, Philipp, Wilhelm Tegethoff, and Jürgen Köhler (2017). “Method for designing waste heat recovery systems (WHRS) in vehicles considering optimal control”. In: *Energy Procedia* 129, pp. 232–239.
- Pfeiffer, Andreas (2008). “Numerische Sensitivitätsanalyse unstetiger multidisziplinärer Modelle mit Anwendungen in der gradientenbasierten Optimierung”. PhD thesis. Martin-Luther Universität Halle-Wittenberg.
- Psichogios, Dimitris C. and Lyle H. Ungar (1992). “A hybrid neural network-first principles approach to process modeling”. In: *AIChE Journal* 38.10, pp. 1499–1511. DOI: <https://doi.org/10.1002/aic.690381003>.
- Rackauckas, Christopher, Yingbo Ma, et al. (2021). *Universal Differential Equations for Scientific Machine Learning*. arXiv: 2001.04385 [cs.LG].
- Rackauckas, Christopher and Qing Nie (2017). “DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia”. In: *The Journal of Open Research Software* 5.1. DOI: 10.5334/jors.151.
- Sapienza, Facundo et al. (2024). *Differentiable Programming for Differential Equations: A Review*. arXiv: 2406.09699 [math.NA]. URL: <https://arxiv.org/abs/2406.09699>.
- Serban, Radu and Alan C Hindmarsh (2003). *CVODES: An ODE solver with sensitivity analysis capabilities*. Tech. rep. Citeseer.
- Thummerer, Tobias, Fabian Jarmolowitz, et al. (2025). “Br(e)aking the boundaries of physical simulation models: Neural Functional Mock-up Units for Modeling the Automotive Braking System”. In: *16th International Modelica & FMI Conference, Lucerne, Switzerland, Sep 8-10, 2025*. (submitted).
- Thummerer, Tobias, Lars Mikelsons, and Josef Kircher (2021). “NeuralFMU: towards structural integration of FMUs into neural networks”. In: *Proceedings of 14th Modelica Conference 2021, Linköping, Sweden, September 20-24, 2021*. Ed. by Martin Sjölund et al. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181297.
- Thummerer, Tobias, Johannes Stoljar, and Lars Mikelsons (2022). “NeuralFMU: Presenting a Workflow for Integrating Hybrid NeuralODEs into Real-World Applications”. In: *Electronics* 11.19. ISSN: 2079-9292. DOI: 10.3390/electronics11193202. URL: <https://www.mdpi.com/2079-9292/11/19/3202>.
- Thummerer, Tobias, Johannes Tintenherr, and Lars Mikelsons (2021-11). “Hybrid modeling of the human cardiovascular system using NeuralFMUs”. In: *Journal of Physics: Conference Series* 2090.1, p. 012155. DOI: 10.1088/1742-6596/2090/1/012155.



# Hybrid Simulation Models for Embedded Applications: A Modelica and eFMI approach

Tobias Kamp<sup>1</sup> Christoff Bürger<sup>2</sup> Johannes Rein<sup>1</sup> Jonathan Brembeck<sup>1</sup>

<sup>1</sup>Institute of Vehicle Concepts, German Aerospace Center, {tobias.kamp, johannes.rein, jonathan.brembeck}@dlr.de

<sup>2</sup>Dassault Systèmes AB, Sweden, christoff.buerger@3ds.com

## Abstract

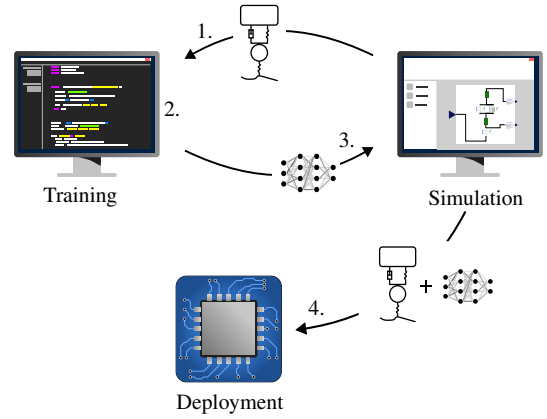
Hybrid simulation models combine physics equations with trainable components to improve simulation results and performance. Physics-enhanced neural ordinary differential equations (PeN-ODE) are a promising type of hybrid models that combine artificial neural networks (NN) with the differential equations of a dynamic system. Dynamic simulation models are often part of embedded control algorithms of cyber-physical systems (CPS); compliance with the safety and real-time requirements of such embedded environments is, however, challenging.

In this work, we propose a workflow to incorporate trained NNs in Modelica models to form hybrid simulation models that are PeN-ODEs. We thereby focus on the transformation steps from equation-based trained PeN-ODEs in Modelica towards causal solutions suited for the embedded domain – up to and including MISRA C:2023 compliance checks and final software-in-the-loop (SiL) tests of generated production code in the modeling environment – for which we leverage eFMI standard compliant tools (Dymola and Software Production Engineering). It is of particular interest how the trained NNs of the hybrid model are implemented. We present two approaches: (1) generation of C code using existing Open Neural Network Exchange (ONNX) tooling and (2) pure Modelica code with the tensor-flow represented as multi-dimensional equations. Both approaches are discussed, highlighting why (2) is, in the long run, a better option given the eFMI technology space.

**Keywords:** Hybrid modeling, neural network, machine learning, embedded system, Modelica, eFMI, Physics-enhanced Neural ODE, recalibration

## 1 Introduction

Two important objectives for system engineers that try to improve their simulation models are (1) minimizing the simulation-to-reality gap and (2) optimizing the model with respect to the simulation performance. In addition to traditional methods, machine learning (ML) methods offer varying data-driven approaches (Rai and Sahu 2020). Addressing the first objective in this context typically means using ML methods to learn (missing) dynamics from real-world measurements, thereby enhancing the predictive ac-



**Figure 1.** Development workflow of hybrid simulation models, including final deployment on the embedded target of a cyber-physical system.

curacy. To approach the second objective, surrogate models can be trained to replace computationally expensive components, using these very components to generate the necessary training data.

It has been shown that dynamical systems can be thoroughly represented by recurrent neural networks (RNN) (Funahashi and Nakamura 1993), physics-informed neural networks (PINN) (Raissi, Perdikaris, and Karniadakis 2019; Cuomo et al. 2022), or neural ordinary differential equations (NODE) (Chen et al. 2018). Hybrid models (Willard et al. 2022) that directly combine trainable components with physics equations usually yield better stability, data efficiency and interpretability. In this study, we focus on a type of hybrid model known as physics-enhanced neural ordinary differential equation (PeN-ODE). PeN-ODEs combine ordinary differential equations (ODE) and artificial neural networks (NN) in a meaningful way (Kamp, Ultsch, and Brembeck 2023). The NNs are trained within the model-equations to capture missing non-linear effects or quantities to improve the predictive quality of the model. The implementation of PeN-ODEs is relatively straightforward, as it requires only the extension of the right-hand side of the ODE-system  $\dot{x}(t) = f(x, u, t)$  with NNs. For the training and simulation of PeN-ODEs, well-established numerical integrator algorithms can be used.

Figure 1 sketches the general development workflow of hybrid models for system simulation, eventually deployed in a cyber-physical system (CPS). Assuming an equation-based model in a simulation environment, such as a Modelica<sup>1</sup> model in Dymola<sup>2</sup>, is already available, the process is covered by Steps 1-4:

**Step 1 (Export):** The physics equations of the model are exported into an ML training environment (e.g., PyTorch<sup>3</sup> in Python<sup>4</sup> or Julia<sup>5</sup>).

**Step 2 (Extension & training):** The physics equations are extended with NNs to form a hybrid model (the PeN-ODE) that is typically trained using gradient-based optimization methods.

**Step 3 (Re-import and simulation):** The trained PeN-ODE or the NNs are re-imported into the simulation environment, enabling simulative validation and combination with other, non-hybrid model parts and/or control algorithms of the CPS.

**Step 4 (Embedded application):** The PeN-ODE is transformed into an implementation suited for deployment on an embedded target of the CPS.

In this study, we assume that the PeN-ODE is already trained – i.e., Steps 1-2 have been accomplished – for example using techniques presented by Thummerer et al. (2022). Thus, we present solutions for Steps 3-4, i.e., the incorporation of trained NNs in Modelica models with the ultimate goal to apply the whole PeN-ODE on an embedded system. The latter objective (Step 4) typically comprises compliance with complex, non-functional embedded domain requirements such as:

- 4a:** MISRA:C 2023 compliance (The MISRA Consortium 2023).
- 4b:** Restricted dependencies on libraries and frameworks.
- 4c:** Worst execution-time and memory-consumption guarantees.
- 4d:** Self-dependent implementation of the PeN-ODE, with its ODEs inline integrated to a level where only linear solver calls are required; such extensive inline integration is typically required to achieve (4b) and (4c), since non-linear solvers jeopardize (4b) and likely violate (4c).
- 4e:** Strict error-handling concepts, especially in case of unexpected Positive or Negative Infinity and Not-a-Number (NaN) results of floating-point operations (IEEE 2019).

**4f:** Software-, processor- and hardware-in-the-loop (SiL, PiL & HiL) tests which are ideally derived from model-in-the-loop (MiL) tests defined in the simulation environment.

To comply with these requirements is of uttermost importance for CPS applications and especially relevant in control engineering, where Step 4 often is the ultimate objective for hybrid models. A typical use-case is to obtain a reduced order/surrogate model for unknown physics or to achieve acceptable performance in an embedded environment where computational resources are scarce.

An important question with respect to Step 4 is how the trained hybrid model – i.e., the PeN-ODE as the source for the embedded solution – looks like. The training and re-import of Steps 2-3 do not necessarily yield a model with the same abstraction level as the original equation-based model. For example, the approach of Thummerer et al. (2022) relies on the Functional Mock-up Interface<sup>6</sup> (FMI) for training and re-import. However, a Functional Mock-up Unit (FMU) is binary code or C source code. This means that the original physics equations are causalized and thus no longer explicitly available in the PeN-ODE. If only the NN parts of the PeN-ODE are re-imported as FMUs, manual integration with the original physics of the model is required. In any case, the tensor-flows of the NNs are hidden inside the FMU.

Thus, starting from an equation-based Modelica model, Steps 2-3 may yield a model with lower abstraction levels than the acausal equations. This holds also for approaches that use the Open Neural Network Exchange<sup>7</sup> (ONNX) format for trained NNs, or any low-level code implementation derived from that. Although re-importing such an implementation into an equation-based Modelica model for the sole purpose of simulation usually poses no issue, e.g., via FMI or the approach presented in Section 3, generating embedded code for the *whole* hybrid model is troublesome because the non-equation parts hamper compliance with requirements 4b-e. In particular, the ODE inline integration for the PeN-ODE (4d) is challenging, but also fulfilling requirement 4e is difficult without equation knowledge. If, on the other hand, Steps 2-3 yield a trained hybrid model whose physics and NNs are modeled as equations or ODEs, the simulation tool can leverage its existing numeric and symbolic manipulations and optimizations to find an algorithmic, causalized inline integration with only linear solver calls.

In the following, we investigate how the tensor-flow of trained PeN-ODEs can be preserved as equations in Step 3, such that Step 4 can leverage existing code generation facilities of the equation-based simulation tooling, which yields a toolchain from acausal physics equations with trained NNs down to safety-critical, hard real-time capable embedded code satisfying 4a-f. The key enabler for Step 4 is the Functional Mock-up Interface for em-

<sup>1</sup><https://modelica.org/language>

<sup>2</sup><https://www.dymola.com>

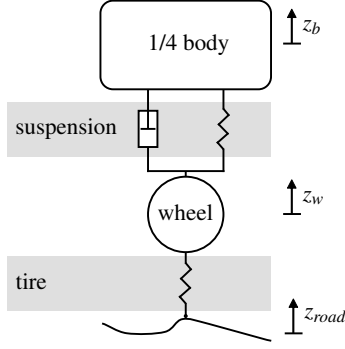
<sup>3</sup><https://pytorch.org>

<sup>4</sup><https://www.python.org>

<sup>5</sup><https://julialang.org>

<sup>6</sup><https://fmi-standard.org>

<sup>7</sup><https://onnx.ai>



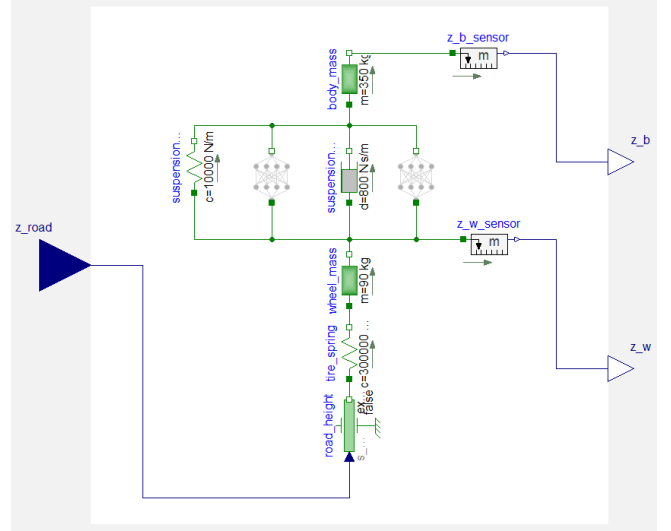
**Figure 2.** Scheme of the simple quarter vehicle model (QVM) comprising the body and wheel masses, a spring to represent the tire and a spring-damper pair for the suspension. The model input is the road height  $z_{road}$ ; the vertical wheel and body positions  $z_w$  and  $z_b$  are the model outputs.

bedded systems<sup>8</sup> (eFMI). The final toolchain relies on the Dymola Modelica tool for equation-based physics modeling and simulation (Step 1), PyTorch for PeN-ODE training (Step 2), and the eFMI support of Dymola and Software Production Engineering<sup>9</sup> for embedded code generation (Step 4). In order to re-import the tensor-flows of trained PeN-ODEs as equations in Dymola (Step 3), we developed a simple Modelica code generator leveraging the open source NeuralNetwork<sup>10</sup> Modelica library.

The rest of the paper is organized as follows: Section 2 presents a motivating case-study which requires the deployment of a trained PeN-ODE on an embedded target and recalibration of NN parameters during runtime. Section 3 presents the C code generation from ONNX models via onnx2c<sup>11</sup> as a first approach for Steps 3-4. Sections 4 and 5 investigate our final solution for Steps 3-4, i.e., a Python to Modelica generator and eFMI with Dymola and Software Production Engineering. Section 6 presents future work, most importantly avoidance of scalarization of tensor-flows in embedded code, and Section 7 finally summarizes the related work regarding hybrid models and embedded code generation in Modelica.

## 2 Quarter vehicle model case-study

We showcase the suitability of the proposed toolchain for embedded code generation of PeN-ODEs (Step 4, a-f) by conducting a case-study for a quarter vehicle model (QVM) that represents the vertical driving dynamics of a road vehicle. QVMs are commonly used in the domain of vehicle dynamics to represent the dynamics of the suspension for controller synthesis and as prediction models



**Figure 3.** Modelica model of the neural QVM, i.e., the PeN-ODE. The two trained NNs capture non-linear effects of the suspension's spring and damper and are connected in parallel to the linear physics components. The NNs can be imported using generated C code from an ONNX representation (cf. Section 3) or as native Modelica equations obtained through a custom Python to Modelica generator (Section 4).

(Fleps-Dezasse and Brembeck 2013; Ultsch, Ruggaber, et al. 2021). Typical applications of QVMs are fault detection of the suspension and the wheels or serving as part of a virtual sensor or as prediction model of controlled, semi-active suspensions that improve driving comfort or road-holding properties (Ultsch, Pfeiffer, et al. 2024). A QVM is limited to the one-dimensional (vertical) dynamics of one wheel and one quarter of the car-body, as depicted in Figure 2. Capturing the vertical dynamics of a road vehicle correctly is challenging, which is due to the influence of non-linear effects such as friction and elasticities (i.e., rubber bushings). To capture such effects, we use a PeN-ODE approach and integrate NNs into the physics equations to learn unknown non-linearities from measurement data, cf. (Kamp, Ultsch, and Brembeck 2023).

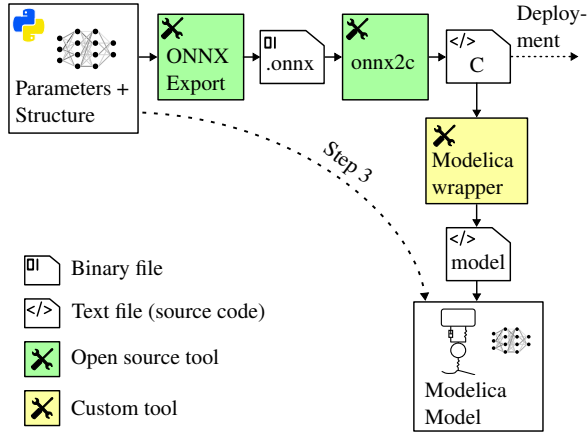
The starting point of the PeN-ODE development is a plain physics-based Modelica model which implements only the linear suspension dynamics. This linear QVM is exported as an FMU and imported into the training environment PyTorch. After extending the differential equations with two NNs – one amending the linear equations of the damper with its non-linear behavior and likewise another for the spring – a gradient-based optimization is conducted. After the training, the trained NNs have to be integrated into the (still linear) Modelica model (Step 3) to obtain a Modelica implementation of the PeN-ODE (cf. Figure 3). In Sections 3 and 4 we present two approaches to accomplish the transfer from the training environment back to the simulation environment. The neural QVM (nQVM) is then used for simulative validation, e.g., using Dymola, and shall further be deployed on an real-time target in the vehicle. There it serves as a prediction model

<sup>8</sup><https://www.efmi-standard.org>

<sup>9</sup><https://my.3dexperience.3ds.com/welcome/compass-world/3dexperience-industries/transportation-and-mobility/smart-safe-and-connected/embedded-software-engineering/systems-software-production-engineer>

<sup>10</sup><https://github.com/AMIT-HSBI/NeuralNetwork>

<sup>11</sup><https://github.com/kraiskil/onnx2c>



**Figure 4.** Scheme of the export toolchain using onnx2c to generate C code for ONNX models. Since many ML-frameworks already support ONNX export, we mark the ONNX export as openly available. Although the code onnx2c generates is suited for embedded application, there is no "ready-to-use" solution for embedded code generation of the *whole* PeN-ODE including its physics equations. The generated C code is imported into Modelica by using a Modelica external C function wrapped into a MSL MIMO block. Listings 1-2 show the wrapper code as generated by our custom tool.

for a control algorithm of the semi-active suspension. In addition, we want to enable the recalibration of the trained NNs during runtime on the embedded system, i.e., without recompilation. This is especially useful when multiple variants of the PeN-ODE are trained to consider changing circumstances, e.g., the current road type or suspension adjustments, as it is common practice in scheduled control algorithms. The application on the embedded target leverages the eFMI standard with tunable NN parameters, which is described in Section 5.

This work captures intermediate results of ongoing work, thus the scope of our case-study is up to the SiL validation of the embedded solution, including the recalibration. The SiL tests allow the comparison with the (continuous) simulation results and constitute the foundation for future HiL and driving tests under real world conditions.

### 3 NNs as external C code

In this section, we present a method to leverage ONNX as a model exchange standard and an open source ONNX to C compiler to incorporate trained NNs in Modelica models. The presented workflow can be automated to a high degree which we underline with generated code samples. However, the dependency on external C code hinders the application of the obtained PeN-ODE on embedded systems through eFMI, since integration with the embedded code generation for the physics part of the PeN-ODE is not obvious. Nevertheless, the method yields plain C code for the NNs that is generally suited for embedded application and is therefore relevant for cases without additional Modelica physics. We will outline how this approach can

be used to deploy trained NNs in a CPS context, even if it is not the favorable approach to our use-case.

#### 3.1 The ONNX format

"ONNX is an open format built to represent machine learning models. ONNX defines a common set of operators – the building blocks of machine learning and deep learning models ..."<sup>7</sup>. Once the model is defined using these operators, ONNX uses the (binary) Protocol Buffers format for serialization. Such ONNX files serve as exchange containers and can be compared to FMUs. The ONNX format is already widely established as an exchange standard for ML models and is supported e.g., by the Keras<sup>12</sup> and PyTorch frameworks, MATLAB®<sup>13</sup> and many more. The ONNX Runtime enables direct inference using an ONNX model.

#### 3.2 ONNX to C compiler

In some applications, relying on the ONNX Runtime is not a valid option, especially when the model should be deployed on an embedded system. The open source onnx2c<sup>11</sup> compiler offers a remedy for users that can make use of plain C code. It is optimized for TinyML<sup>14</sup>, i.e., running on microcontrollers. Provided with an ONNX model, onnx2c generates a single C file with all required functions that correspond to the atomic ONNX operators. The parameter arrays that hold the trainable parameters are likewise contained in the generated code. It has to be mentioned that onnx2c does not support all available operators and not all model typologies and is a privately maintained project. The C code contains a function

```
entry(const float x[1][M], float y[1][N])
```

with the input dimension  $M$  and output dimension  $N$ . This function is the only function that needs to be called in order to perform a forward pass through the model.

#### 3.3 Integration in Modelica

Once the NN is exported as an ONNX model and compiled into C code, it can be used in Modelica using its external C interface (cf. Figure 4). Listing 1 presents an example implementation of the entry-function call that can be used as a template for generation. Further, it can be convenient to wrap the function into a Modelica Standard Library (MSL) MIMO block as shown in Listing 2.

**Listing 1.** Modelica function that calls the entry function from the generated C file (called nn.c). The dimensions  $M$  and  $N$  correspond with the number of in- and outputs of the NN (to be replaced with size\_t values).

```
function call_entry "call_entry"
  input Real u[M];
  output Real y[N];
  external "C" call_nn(u, y)
  annotation(
    IncludeDirectory="<<nn.c dir>>",
```

<sup>12</sup><https://keras.io>

<sup>13</sup><https://www.mathworks.com/products/matlab.html>

<sup>14</sup><https://github.com/mit-han-lab/tinyml>

```

Include="
#include \"nn.c\"
void call_nn(const double* u, double* y)
{
    float u_buf[1][M], y_buf[1][N];
    size_t i;

    for (i = 0; i < M; i++)
    {
        u_buf[0][i] = (float) u[i];
    }

    entry(u_buf, y_buf);

    for (i = 0; i < N; i++)
    {
        y[i] = (double) y_buf[0][i];
    }
}");
end call_entry;

```

**Listing 2.** MIMO wrapper for C-function call (cf. Listing 1); The dimensions M and N correspond with the number of in- and outputs of the NN (to be replaced with Integer values).

```

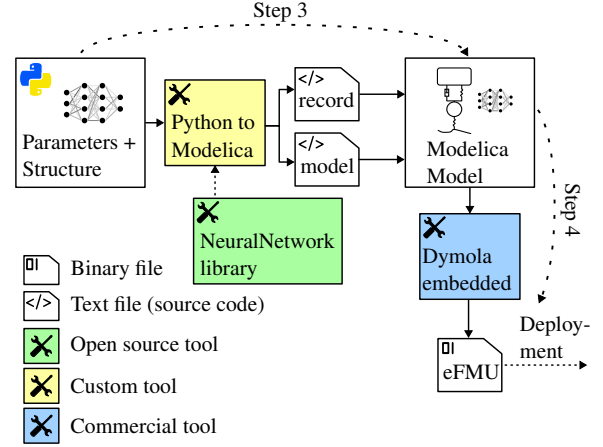
model MyNN
    extends Modelica.Blocks.Interfaces.MIMO(
        nin = M,
        nout = N);
equation
    y = call_entry(u);
end MyNN;

```

If a PeN-ODE contains multiple NNs, one must assure that variable- and function names in the C code are unique. When using onnx2c, this is generally not the case, since the C files are generated independently and at least the entry-function always has the same name. As of now, this can only be solved by editing the C code after its generation. Issues can also occur, when the ONNX model is composed in a way that the operator-nodes have ambiguous or repeating names. This can be avoided by giving a unique name to each node during the ONNX export of the ML model. The node names of an existing ONNX model can still be adapted, e.g., using the ONNX Python API.

### 3.4 Discussion

The presented approach uses the ONNX open standard and an open source ONNX to C compiler. The incorporation of the generated C code in Modelica (Step 3) is straightforward and can be effectively automated by generating corresponding Modelica wrappers. Furthermore, ONNX is a well established exchange standard for ML models and many ML frameworks already offer the export to ONNX. The onnx2c compiler is stable, maintained, and supports many of the existing ONNX operators. It generates C code which is suitable for the application on micro-controllers, which is important for CPSs. In comparison to solutions that use the ONNX Runtime, it has the advantage to be self-contained without any third party library or framework dependencies, which facilitates exchange and portability. Modelica tools can, for example, export PeN-



**Figure 5.** Scheme of the export toolchain employing the Neural-Network library. The Python to Modelica generator is a custom tool that allows direct Modelica code generation from Python. This approach leverages Dymola’s existing symbolic facilities to find algorithmic solutions suited for embedded application and enable embedded code generation for the *whole* PeN-ODE using the open eFMI standard.

ODE models as binary code FMUs without further dependencies.

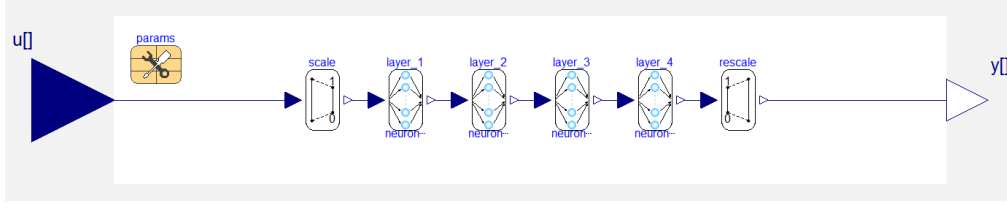
However, the onnx2c approach yields embedded code only for the NN parts of the PeN-ODE. Embedded code generation for the whole PeN-ODE (Step 4) is not straightforward because the production code for the physics equations must be generated and connected with the NNs. This is not a trivial system integration task, since knowledge about causalization, equation system properties and connectivity of the *whole* PeN-ODE must be properly incorporated. There might be, for example, algebraic loops between physics and NNs, or the physics equations and NNs form a mixed system of equations where Boolean conditions depending on NN outputs switch physics behavior fed into the NN. In such cases, the NN code has to be properly embedded into the iterations of the inline integrated ODE solver when generating embedded code for the physics parts of the Modelica model.

These embedded code generation issues for PeN-ODEs can be bypassed by handing the challenges over to the already existing, advanced symbolic facilities of Modelica compilers like Dymola. Section 4 presents an approach for Step 3 where the NN parts of the PeN-ODE are re-imported as pure Modelica equations such that the Modelica compiler can take care of proper integration.

## 4 NNs as native Modelica

In this section, we present an approach that implements trained NNs as Modelica models utilizing the Neural-Network Modelica library. This method allows seamless integration of the NNs with the physics equations and enables eFMI-based embedded code generation for the whole PeN-ODE. Figure 5 summarizes the tooling of this Modelica-centric workflow, where the trained NNs are





**Figure 6.** Generated, equation-based Modelica model of one of the trained NNs of the nQVM, showing its diagrammatic tensor flow using NeuralNetwork library components. The NN parameters (weights and biases) for each of the four dense layers ( $16 \times 1 - 16 \times 16 - 16 \times 16 - 1 \times 16$ ) are contained in the parameter record that can be marked for exposure as tunable eFMI parameters.

transformed back to Modelica equations.

#### 4.1 The NeuralNetwork library

The NeuralNetwork Modelica library was first published in 2006 (Codecà and Casella 2006). Since 2023, it is further developed by the University of Applied Sciences and Arts Bielefeld (HSBI) under open source 3-Clause BSD Licence<sup>10</sup>. As of version 2.1, it offers implementations to compose multi-layer perceptrons (MLP) including dense layers, scaling, standardizing, and principal component analysis (PCA) layers, and the activation functions rectified linear unit (ReLU), sigmoid, hyperbolic tangent (tanh), softplus and unit step.

Considering the vast variety of ML architectures, this is only a basic set. For PeN-ODEs however, simple MLPs suffice in many cases (Thummerer, Stoljar, and Mikelsons 2022; Kamp, Ultsch, and Brembeck 2023).

#### 4.2 Generation of Modelica NNs

We implemented a custom Python to Modelica code generator to automate the implementation of trained NNs using the definitions of the NeuralNetwork library. Listing 3 presents an example of a simple MLP that was generated in this manner. We deem it best to generate a separate Modelica record alongside the NN that holds its parameters, i.e., NN weights and biases (cf. Listing 4). This record can be used to redeclare the parameters of the NN block and thus enables a fast exchange. This is useful when multiple variants of the NN are to be validated in simulation, given that the NN architecture (number of layers, number of neurons, and activation functions) is maintained. Figure 6 shows the composition of different layers of a typical MLP with the corresponding parameter record.

**Listing 3.** Sample of a generated Modelica NN. The NeuralNetwork library components are wrapped into a MIMO block comparable to the external C approach (cf. Listing 2).

```
block MyNN
  extends Modelica.Blocks.Interfaces.MIMO;
  import NN = NeuralNetwork;
  // NN parameter record
  parameter MyParametrization params;
  // NN layers
  NN.Layer.Dense layer_1(
    weights = params.layer_1_weights,
    bias = params.layer_1_bias,
    redeclare function f =
      NN.ActivationFunctions.ReLU);
```

```
NN.Layer.Dense layer_2(
  weights = params.layer_2_weights,
  bias = params.layer_2_bias,
  redeclare function f =
    NN.ActivationFunctions.Id);
```

```
equation
  // NN composition
  connect(u, layer_1.u);
  connect(layer_1.y, layer_2.u);
  connect(layer_2.y, y);
end MyNN;
```

**Listing 4.** Generated parameter record defining default weights and biases for the NN of Listing 3, exposed as tunable eFMI parameters for the embedded code generation of Section 5.

```
record MyParametrization
  extends Modelica.Icons.Record;
  // Trained NN parameters
  parameter Real [16, 1] layer_1_weights =
    {{2.15}, {-0.713}, ...};
  parameter Real [16] layer_1_bias =
    {0.174, -0.0312, ...};
  parameter Real [1, 16] layer_2_weights =
    {{0.215, 1.342, ...}};
  parameter Real [1] layer_2_bias =
    {0.230};
  // Expose as tunable eFMI parameters
  annotation (
    __Dymola_eFMI_ExposeTunableParameters =
      true);
end MyParametrization;
```

#### 4.3 Discussion

The pure Modelica approach offers a seamless integration of trained NNs with the physics part of the PeN-ODE in Modelica. The NN parameters can be encapsulated in records separated from the NN architecture, facilitating variation and training validation. A drawback compared to the usage of ONNX via onnx2c is the limited number of available architectures of the NeuralNetwork library. Further, a complete Python to Modelica generator leveraging the NeuralNetwork library is not (yet) freely available. Our implementation is, as of now, just a functional prototype and not published. The most important advantage of the native Modelica approach is the possibility to export the *whole* PeN-ODE via eFMI for embedded application as motivated and described in the following Section 5.

## 5 Embedded code via eFMI

In this section, we motivate the use of the eFMI standard for Modelica equation-based hybrid models in embedded applications. Our solution relies on the import of trained NNs as Modelica equations as presented in Section 4, meets requirements 4a-f of Section 1, and enables online recalibration of NN parameters.

### 5.1 The eFMI Standard

The Functional Mock-up Interface for embedded systems (eFMI) is "an open standard for the ... model-transformation-based development of advanced control functions suited for safety-critical and real time targets ... [, defining a] container architecture ... from high-level modeling and simulation – e.g., a-causal, equation-based physics in Modelica – down to actual embedded code"<sup>8</sup>.

The key interface between the physics simulation and embedded application domains is the Guarded Algorithmic Language for Embedded Control (GALEC). To deploy an equation-based simulation model in a safety-critical, hard real-time environment, the simulation tool has to transform the model into a GALEC program (Algorithm Code container) and hand it over to any eFMI-aware production code generator for final embedded code generation (Production and Binary Code container). The idea is, that each tooling can be a domain expert on its abstraction-level without bothering about the other domains. A Modelica tool like Dymola knows a lot about discretization, symbolic optimization and inline integration – i.e., how to find a sampled algorithmic solution expressible in GALEC – whereas production code generators like Software Production Engineering are knowledgeable about satisfying embedded domain coding requirements but have no concept of modeling physics for equation-based simulation.

GALEC has some language characteristics making it a convenient intermediate representation between modeling and embedded software (Lenord et al. 2021). It is target independent, supports multi-dimensional arithmetic, and a rich set of built-in functions to abstract common math operations including interpolation and solving systems of linear equations. Most importantly, it is computational-safe, which means that a powerful static evaluation concept is used to guarantee all indexing is within bounds, all loops have an upper-bound, and all potential runtime errors like NaNs, singular linear equation systems or domain errors of built-in functions (like poles of trigonometric functions) are handled or explicitly exposed to the runtime environment. GALEC programs satisfy 4b-e of Step 4 by definition. Once a simulation tool finds a sampled solution for the simulation problem – which is not always possible for general equation systems – and expresses it as GALEC program, further tooling towards embedded C code "only" has to preserve these characteristics and can focus on code optimization (e.g., loop and multi-dimensional expression unrolling) and embedded target environment compliance (e.g., compliance with style guidelines, code analysers, in-

terfaces or system integration requirements).

Besides Algorithm, Production and Binary Code containers, eFMI also defines Behavioral Model containers to define test-scenarios. Typically, Behavioral Models are derived from continuous MiL experiments in a physics simulation environment and shared for later SiL and HiL tests of production code.

### 5.2 eFMI support in Dymola

Dassault Systèmes provides several tools supporting eFMI: Dymola for generating eFMI Algorithm Code containers for synchronous Modelica models, Software Production Engineering for generating eFMI Production, and in turn, Binary Code containers and AUTOSAR Builder<sup>15</sup> to extend Production Code containers to become AUTOSAR<sup>16</sup> components.

It is noteworthy, that Software Production Engineering is also used as embedded code generation backend for No Magic Cameo Systems Modeler<sup>17</sup>, a development environment for model-driven engineering, system architecture, system design, and software engineering based on SysML<sup>18</sup>. Generated production code therefore is of industry-level quality, for example MISRA C:2023 compliant. The integration with Dymola is seamless. Dymola provides facilities to configure 32 bit and 64 bit floating-point precision for production code generation, import production code for SiL simulation via generated Modelica wrappers, derive SiL tests from existing MiL experiments, check MISRA C:2023 compliance with Cppcheck Premium<sup>19</sup>, import production code in MATLAB®/Simulink®<sup>20</sup> as C Function blocks, and export eFMUs with deployment-ready production code as FMUs.

### 5.3 QVM case-study with Dymola

Based on Dymola's eFMI support presented in the previous section, generating embedded code for the nQVM of Section 2, with its damper and spring NNs represented as NeuronalNetwork library models according to Section 4, has been straightforward. The two major challenges were, first, how to conveniently expose the NN weights and biases as tunable eFMI parameters, and second, how to model the respective recalibration SiL tests in Modelica.

For the first issue, we extended Dymola with a new annotation for Modelica records. Adding `__Dymola_eFMI_ExposeTunableParameters = true` to a record *class* marks the independent parameters of *instances* for exposure as tunable eFMI parameters, regardless how deeply nested within models they are. The Python to Modelica generator of Section 4 can hence automatically add the annotation to the NN parametrization record (cf. Listing 4).

<sup>15</sup><https://www.3ds.com/products/catia/autosar-builder>

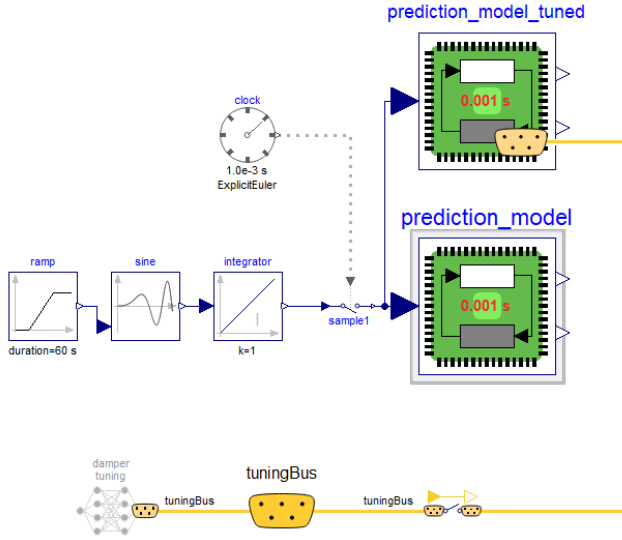
<sup>16</sup><https://www.autosar.org>

<sup>17</sup><https://www.3ds.com/products/catia/no-magic>

<sup>18</sup><https://www.omg.sysml.org>

<sup>19</sup><https://www.cppcheck.com>

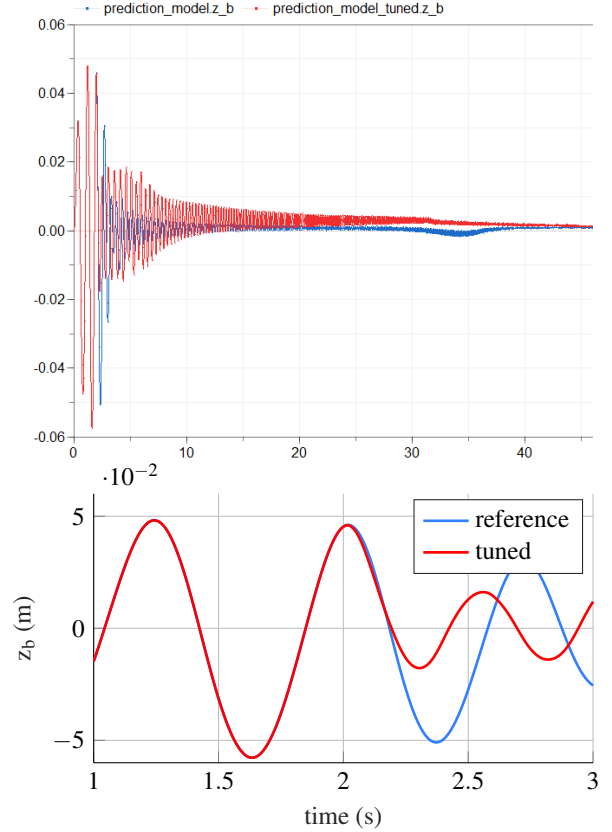
<sup>20</sup><https://www.mathworks.com/products/simulink.html>



**Figure 7.** Recalibration SiL test in Dymola: The nQVM PeN-ODE is instantiated twice – once untuned and once tuned – via a generated Modelica wrapper backed by the eFMUs production code. The damper tuning component is manually implemented. It triggers recalibration at 2 s and applies updated NN parameters for the learned friction model on the generated tuning bus.

The second issue, how to model recalibration SiL tests, exposes a general Modelica shortcoming. From the eFMI perspective, recalibration test scenarios are well-supported in Behavioral Model containers; and Dymola supports the derivation of Behavioral Models from existing Modelica experiments by searching them for instances of the model that is subject to eFMI code generation. Each of those instances becomes a SiL test scenario, where the instance inputs are the stimuli and the outputs the reference trajectories. This concept works well to automatically derive SiL test suites from ordinary Modelica experiments. However, Modelica has no online recalibration concept. In Modelica, parameters can be only modified before, but not during the simulation. Hence, online recalibration must be modeled by runtime values encoding "parameters". Since this implies extensive model changes on the models to recalibrate, recalibration support for ordinary Modelica models is pointless. In our case however, we can provide recalibration facilities in the Modelica wrappers Dymola generates for SiL simulation of production code generated by Software Production Engineering. To this end, the wrappers provide an optional tuning bus, which can be enabled whenever recalibration shall be tested. Actual calibrations are simply connected to the bus, requiring manual modeling of a continuous time-based source provisioning the parameter sets to apply at each time point.

Figure 7 shows the recalibration scenario we used to validate the nQVM. In this experiment, we recalibrate the damper-NN with a slightly altered (learned) friction model after 2 s. The PeN-ODE is simulated with a chirp signal (sine-sweep) as input. In Figure 8, the simulation result



**Figure 8.** SiL simulation results of the body height  $z_b$  for the untuned (blue) and tuned (red) nQVM. At 2 s simulation time, the recalibration is triggered and leads to a new system behavior.

of the body height  $z_b$  of the tuned and untuned PeN-ODE is plotted. Although the tuning clearly alters the dynamics of the system, the state of the PeN-ODE is not changed when recalibrating such that there are no discontinuities in the trajectory. If recalibration would take several sampling steps, which can be expected when using actual real-time hardware, the current state would be held – i.e., no samplings are conducted – until the recalibration is terminated. Nevertheless, it is likely that observers or control algorithms using the PeN-ODE converge much faster when continuing sampling from the hold state compared to a full state reset. Although the nQVM is stable in the recalibration scenario of Figures 7 and 8, no such guarantee can be given for general equation systems; if, and under which constraints, an independent parameter indeed is suited for online tuning, is a domain specific characteristic.

## 6 Future work

This paper presents intermediate results of the ongoing research in the PeN-ODE-related work package (WP4) of the ITEA 4 Project OpenSCALING<sup>21</sup> (cf. Acknowledgements). The work scheduled for the second half of the project ending in October 2026 can be separated into three tasks, ranked from highest to lowest priority: (1) preserva-

<sup>21</sup><https://itea4.org/project/openscaling.html>



tion of the multi-dimensional arithmetic of the tensor-flow of NNs in production code, (2) actual system integration of the eFMU of the case-study presented in Section 2 with HiL and driving tests under real-world conditions including recalibration of NN parameters and (3) support of the System Structure & Parametrization<sup>22</sup> (SSP) standard for NN parametrization and recalibration.

## 6.1 Multi-dimensional embedded tensor-flows

Most Modelica tools, including Dymola, transform models into an index 1 differential algebraic equation (DAE) system, where each element of multi-dimensions is scalarized to individual variables; doing so, multi-dimensional arithmetic operations are flattened to a set of individual scalar operations (scalarization). Although this representation is essential for many advanced numeric and symbolic manipulations and optimizations, it also significantly increases code size when scalarizing the tensor-flows of the equation-based NNs presented in Section 4. For embedded applications, code size is critical. Hence, it is important to avoid the scalarization of tensor-flows. However, for the physics-parts of PeN-ODEs it still is beneficial to find symbolic solutions to avoid linear system solver calls.

In our case-study, the actual physics equations of the nQVM only result in ~4 KB of eFMI GALEC code including two symbolically solved size-1 linear equation systems, whereas the scalarized tensor-flows of the spring and damper NNs – each holding 593 parameters in the input layer ( $16 \times 1$ , ReLu), two hidden layers ( $16 \times 16$ , ReLu), and the output layer ( $1 \times 16$ , Identity), each with bias (cf. Figure 6) – produce ~117 KB of code. This is a lot, considering that the tensor-flows of the NNs could be written as single-line multi-dimensional GALEC expressions. The ratios for derived C production and binary code are similar. The onnx2c solution of Section 3, which preserves the tensor-flows as for-loops, only yields ~5.5 KB of C code. All numbers are excluding the initialization of NN weights and biases, which contributes significantly to code size but cannot be avoided.

The prevention of scalarization is subject of work package 3 (WP3), large scale system (LSS) modeling, of the OpenSCALING project. Although the development focuses on physics based LSSs – like electric power grids with homogeneous components of huge quantities modeled as arrays of Modelica components – the planned solutions are likely also applicable in our use-case. If anything, tensor-flows are even simpler to handle because they are clean – i.e., not mixed with exceptions or zero crossing logic – arithmetic. Foundational research on how to preserve multi-dimensional operations for index 1 DAEs without sacrificing symbolic optimizations is already available (Otter and Elmqvist 2017; Abdelhak, Casella, and Bachmann 2023).

## 6.2 eFMU system integration and CPS tests

The presented toolchain has been validated up to, and including, SiL tests of the generated eFMU in Dymola. The actual system integration on a dedicated embedded target – for our case-study this involves the deployment of the nQVM as prediction model on the embedded control system of our experimental research platform (Ruggaber et al. 2023) – still must be conducted. We are confident, that our eFMI approach is well-suited for that eFMUs generated by Dymola and Software Production Engineering have been successfully system-integrated in the past, including final CPS tests under real-world conditions (Ultsch, Ruggaber, et al. 2021). A new challenge is the online recalibration for tuning the weights of the embedded NNs, which we deem non-critical, considering the successful SiL experiment of the recalibration facilities of the generated eFMU.

## 6.3 SSP standard support

Online recalibration of PeN-ODE NN-weights is related to ongoing SSP standardization efforts. SSP is supposed to be a universal standard for defining varying parameter sets for interconnected FMUs (Hällqvist et al. 2021). Modelica support, in particular SSP import and export in Dymola, is available (Brück 2023). Our tunable parameter record approach of Sections 4 and 5 for modeling NN-weights can be mapped directly using Dymola’s existing SSP facilities. In FMI and eFMI however, recalibration of tunable parameters can be seen as a *timed sequence* of parameter changes; however, SSP has no notion of timing. Likewise, Modelica lacks the means to conveniently model online recalibration. In Modelica, parameters can only be changed by means of static modifications before simulation starts, which is why the SiL tests presented in Section 5 had to implement new calibrations via ordinary Modelica variables instead of parameters. This is troublesome and requires the definition of event-points for recalibration throughout the continuous simulation. To align SSP, Modelica, FMI and eFMI such that recalibration test-scenarios can be modeled as timed SSP parametrizations is an open issue, very much in the scope of OpenSCALING and its objective to harmonize the Modelica Association’s<sup>23</sup> open standards ecosystem.

## 7 Related work

Our previous work provided an extended introduction to PeN-ODEs, including how to define and train them (Kamp, Ultsch, and Brembeck 2023). Requirements 4a-f, to enable embedded and CPS applications, were not considered however.

Thummerer et al. (2022) proposed a solution for export, training, and re-import of PeN-ODEs in simulation environments (Steps 1-3) using the FMI standard. Their approach is feasible even for complex systems, and especially in industrial applications where discretion is of importance. Thanks to the ubiquitous availability of

<sup>22</sup><https://ssp-standard.org>

<sup>23</sup><https://modelica.org/association>

FMI in the simulation domain, and increasing support also in training environments (e.g., FMPy<sup>24</sup> for Python or FMI.jl<sup>25</sup> for Julia), this approach is widely applicable. However, FMUs are implementation black-boxes, hiding the involved equations such that Step 4, preparing PeN-ODEs for embedded application, is not straightforward. The approach does not consider the embedded domain requirements 4a-f.

The open source SMARtInt library<sup>26</sup> offers a way to use ONNX models and TensorFlow<sup>27</sup> models exported as Lite Runtime<sup>28</sup> (LiteRT) in Modelica, supporting a huge range of NN types and architectures. However, its dependency on a plethora of heterogeneous third party frameworks and technologies severely impedes embedded application when compliance with 4b-e is required.

Hübel et al. (2022) trained a surrogate model that is benchmarked inside a Modelica model. Comparable to our approach, they used the NeuralNetwork Modelica library to transfer the trained NN to Modelica. They also mentioned the possibility to use generated C code as a future research topic, but did not elaborate on embedded applications.

In the ITEA EMPHYSIS project (EMPHYSIS International Consortium 2021), which bore the eFMI standard, multiple demonstrators used eFMUs to incorporate NNs into predictive model control applications targeting embedded systems. The starting point of embedded code generation were trained NNs however, not hybrid models. Hence, although 4a-c and 4e-f are supported by representing NNs as eFMI GALEC programs and leveraging the eFMI tooling as we do, the combination with physics equations – and therefore an ODE inline integration to a level where only linear solver calls are required (4d) – has not been investigated.

The work of Ultsch et al. (2021) within the EMPHYSIS project can be regarded as predecessor to our study. Similar to the QVM case-study of Section 2, they transformed a nonlinear prediction model in Modelica via eFMI into an embedded solution suited for the model-based control of the semi-active dampers of a car. An extended Kalman filter was used for the state estimation based on the prediction model. To this end, the eFMU of the prediction model was integrated into a dedicated Kalman filter library implemented in C. The state estimation algorithm of the Kalman filter thereby perturbs discretized continuous states of the prediction model by explicitly setting them before performing an integration step with the inline integrated solver. The eFMU was deployed on the ECU of a series-produced car and validated in driving tests under real-world conditions.

Kurzbach et al. (2023) proposed a low-level equation language for Modelica, in which the higher level

object-oriented concepts like inheritance, modifications and nested components are flattened to simple equations. The motivation for such a "base Modelica" is to facilitate the integration of third party technology spaces with Modelica tooling. If such a representation becomes part of the Modelica standard and would be widely adopted, it could significantly ease bridging the simulation and ML domains (Step 1 and 3 of our approach); in particular, if it could be used as an actual *model* exchange format in FMI. Although the definition of a basic-equation language for Modelica has been a long time vision of the Modelica Association<sup>23</sup>, it is much too early to judge if the proposal will be standardized and sufficiently widely adopted.

## 8 Conclusions

We presented a toolchain for the embedded application of PeN-ODE hybrid models derived from Modelica physics equations. In order to meet the non-functional requirements on embedded software, we relied on the Functional Mock-up Interface for embedded systems (eFMI) and its support in the Modelica tool Dymola. Dymola's existing symbolic optimization and inline-integration routines are prolific in finding causalized, algorithmic solutions for complex equation systems, in particular to avoid non-linear solver calls as required in embedded applications. The export of algorithmic solutions as eFMI GALEC programs in turn enables further eFMI support to eventually derive high-quality production code.

A challenge of this approach is the correct integration of the trained NN-components of PeN-ODEs with the physics part, such that the integrated solvers of embedded solutions can properly handle the interactions between NNs and physics-equations, like event-handling and zero-crossings in mixed systems of equations, smoothness requirements, sampling and discretization, etc. We avoid this open research question by representing the tensor flow as such – i.e., as *multi-dimensional expression* – eventually yielding an equations-only representation of the whole PeN-ODE which can be processed by Dymola's existing symbolic facilities. For the required re-import of the trained NNs into the original physics-equations, we developed a Python to Modelica code generator targeting the NeuralNetwork Modelica library.

We validated our approach with the help of a neural quarter vehicle model (nQVM) case-study. SiL tests of the eFMI solution demonstrate its applicability, including recalibration tests for online changes of NN parameters. Left as future work is to extend Dymola's symbolic facilities such that they avoid scalarization of the tensor-flows of NNs and final system integration of the nQVM with HiL and driving tests under real world conditions.

## Acknowledgements

This work has been supported by the Swedish Agency for Innovation Systems (Vinnova<sup>29</sup>, grant number 2023-

<sup>24</sup><https://github.com/CATIA-Systems/FMPy>

<sup>25</sup><https://github.com/ThummeTo/FMI.jl>

<sup>26</sup><https://github.com/xrg-simulation/SMARtInt>

<sup>27</sup><https://www.tensorflow.org>

<sup>28</sup>Formerly known as TensorFlow Lite (TFLite).

<sup>29</sup><https://www.vinnova.se>




00969) and the German Federal Ministry of Education and Research (BMBF<sup>30</sup>, grant number FKZ 01IS23062A) within the ITEA 4 Project Open standards for SCALable virtual engineeriNG and operation (OpenSCALING<sup>21</sup>, ITEA Project 22013).

## References

- Abdelhak, Karim, Francesco Casella, and Bernhard Bachmann (2023-10). “Pseudo Array Causalization”. In: *Proceedings of the 15th International Modelica Conference*. Vol. 204. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 177–188. DOI: 10.3384/ecp204177.
- Brück, Dag (2023-10). “SSP in a Modelica Environment”. In: *Proceedings of the 15th International Modelica Conference*. Vol. 204. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 711–715. DOI: 10.3384/ecp204711.
- Chen, Ricky T. Q. et al. (2018-12). “Neural Ordinary Differential Equations”. In: *Proceedings of The Thirty-Second Annual Conference on Advances in Neural Information Processing Systems*. Curran Associates, Inc., pp. 6571–6583. DOI: 10.48550/arXiv.1806.07366.
- Codecà, Fabio and Francesco Casella (2006-09). “Neural Network Library in Modelica”. In: *Proceedings of the 5th International Modelica Conference – Volume 2*. Modelica Association, pp. 549–557.
- Cuomo, Salvatore et al. (2022-07). “Scientific Machine Learning Through Physics-Informed Neural Networks: Where we are and What’s Next”. In: *Journal of Scientific Computing* 92.3. DOI: 10.1007/s10915-022-01939-z.
- EMPHYSIS International Consortium (2021-09). *EMPHYSIS – D7.9 eFMI for physics-based ECU controllers – Public report*. Tech. rep. ITEA 3 project 15016. ITEA. URL: <https://www.efmi-standard.org/media/resources/emphysis-public-demonstrator-summary.pdf>.
- Fleps-Dezasse, Michael and Jonathan Brembeck (2013). “Model based vertical dynamics estimation with Modelica and FMI”. In: *IFAC Proceedings Volumes* 46.21, pp. 341–346. DOI: 10.3182/20130904-4-jp-2042.00086.
- Funahashi, Ken-ichi and Yuichi Nakamura (1993-01). “Approximation of dynamical systems by continuous time recurrent neural networks”. In: *Neural Networks* 6.6, pp. 801–806. DOI: 10.1016/s0893-6080(05)80125-x.
- Hällqvist, Robert et al. (2021-09). “Engineering Domain Interoperability Using the System Structure and Parameterization (SSP) Standard”. In: *Proceedings of 14th Modelica Conference 2021*. Vol. 181. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 37–48. DOI: 10.3384/ecp2118137.
- Hübel, Moritz et al. (2022-11). “Hybrid physical-AI based system modeling and simulation approach demonstrated on an automotive fuel cell”. In: *Proceedings of Asian Modelica Conference 2022*. Vol. 193. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 157–163. DOI: 10.3384/ecp193157.
- IEEE (2019-07). *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2019 (Revision of IEEE 754-2008). Institute of Electrical and Electronics Engineers. ISBN: 978-1-5044-5924-2. DOI: 10.1109/IEEESTD.2019.8766229.
- Kamp, Tobias, Johannes Ultsch, and Jonathan Brembeck (2023). “Closing the Sim-to-Real Gap with Physics-Enhanced Neural ODEs”. In: *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics*. SCITEPRESS - Science and Technology Publications. DOI: 10.5220/0012160100003543.
- Kurzbach, Gerd et al. (2023-10). “Design proposal of a standardized Base Modelica language”. In: *Proceedings of the 15th International Modelica Conference*. Vol. 204. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 469–477. DOI: 10.3384/ecp204469.
- Lenord, Oliver et al. (2021-09). “eFMI: An open standard for physical models in embedded software”. In: *Proceedings of 14th Modelica Conference 2021*. Vol. 181. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 57–71. DOI: 10.3384/ecp2118157.
- Otter, Martin and Hilding Elmqvist (2017-05). “Transformation of Differential Algebraic Array Equations to Index One Form”. In: *Proceedings of the 12th International Modelica Conference*. Vol. 132. Linköping Electronic Conference Proceedings. Modelica Association and Linköping University Electronic Press, pp. 565–579. DOI: 10.3384/ecp17132565.
- Rai, Rahul and Chandan K. Sahu (2020). “Driven by Data or Derived Through Physics? A Review of Hybrid Physics Guided Machine Learning Techniques With Cyber-Physical System (CPS) Focus”. In: *IEEE Access* 8, pp. 71050–71073. DOI: 10.1109/access.2020.2987324.
- Raissi, M., P. Perdikaris, and G.E. Karniadakis (2019-02). “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378, pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045.
- Ruggaber, Julian et al. (2023-02). “AI-For-Mobility—A New Research Platform for AI-Based Control Methods”. In: *Applied Sciences* 13.5. DOI: 10.3390/app13052879.
- The MISRA Consortium (2023-04). *MISRA C:2023 – Guidelines for the use of the C language in critical systems*. Third edition, Second revision. The MISRA Consortium Limited. ISBN: 978-1-911700-09-8.
- Thummerer, Tobias, Johannes Stoljar, and Lars Mikelsons (2022-10). “NeuralFMU: Presenting a Workflow for Integrating Hybrid NeuralODEs into Real-World Applications”. In: *Electronics* 11.19, p. 3202. DOI: 10.3390/electronics11193202.
- Ultsch, Johannes, Andreas Pfeiffer, et al. (2024-08). “Reinforcement Learning for Semi-Active Vertical Dynamics Control with Real-World Tests”. In: *Applied Sciences* 14.16. DOI: 10.3390/app14167066.
- Ultsch, Johannes, Julian Ruggaber, et al. (2021-11). “Advanced Controller Development Based on eFMI with Applications to Automotive Vertical Dynamics Control”. In: *Actuators* 10.11. DOI: 10.3390/act10110301.
- Willard, Jared et al. (2022-11). “Integrating Scientific Knowledge with Machine Learning for Engineering and Environmental Systems”. In: *ACM Computing Survey* 55.4. Revised and extended version of “Integrating Physics-Based Modeling With Machine Learning: A Survey”. DOI: 10.1145/3514228.

<sup>30</sup><https://www.bmbf.de>

# Efficient Training of Physics-enhanced Neural ODEs via Direct Collocation and Nonlinear Programming

Linus Langenkamp <sup>1</sup> Philip Hannebohm <sup>1</sup> Bernhard Bachmann <sup>1</sup>

<sup>1</sup>Institute for Data Science Solutions, Bielefeld University of Applied Sciences and Arts, Germany,  
{first.last}@hsbi.de

## Abstract

We propose a novel approach for training Physics-enhanced Neural ODEs (PeN-ODEs) by expressing the training process as a dynamic optimization problem. The full model, including neural components, is discretized using a high-order implicit Runge-Kutta method with flipped Legendre-Gauss-Radau points, resulting in a large-scale nonlinear program (NLP) efficiently solved by state-of-the-art NLP solvers such as Ipopt. This formulation enables simultaneous optimization of network parameters and state trajectories, addressing key limitations of ODE solver-based training in terms of stability, runtime, and accuracy. Extending on a recent direct collocation-based method for Neural ODEs, we generalize to PeN-ODEs, incorporate physical constraints, and present a custom, parallelized, open-source implementation. Benchmarks on a Quarter Vehicle Model and a Van-der-Pol oscillator demonstrate superior accuracy, speed, generalization with smaller networks compared to other training techniques. We also outline a planned integration into OpenModelica to enable accessible training of Neural DAEs.

**Keywords:** *Physics-enhanced Neural ODEs, Dynamic Optimization, Nonlinear Programming, Modelica, Neural ODEs, Universal Differential Equations*

## 1 Introduction

Growing access to real-world data and advances in computational modeling have opened new possibilities for combining measured data with physics-based models. Neural Ordinary Differential Equations (NODEs) (Chen et al. 2018) represent a significant advancement in merging data-driven machine learning with physics-based modeling. By replacing the dynamics of an ODE with a neural network

$$\dot{\mathbf{x}}(t) = NN_{\mathbf{p}}(\mathbf{x}(t), \mathbf{u}(t), t), \quad (1)$$

where  $\mathbf{x}(t)$  are states,  $\mathbf{u}(t)$  is a fixed input vector, and  $\mathbf{p}$  are the neural network parameters, NODEs bridge the gap between traditional differential equations and modern deep learning. After obtaining a NODE and given an initial condition  $\mathbf{x}(t_0) = \mathbf{x}_0$ , the state trajectory is reconstructed by simulation with an arbitrary ODE solver

$$\mathbf{x}(t) := \text{ODESolve}(NN_{\mathbf{p}}(\mathbf{x}(t), \mathbf{u}(t), t), \mathbf{x}_0). \quad (2)$$

However, learning the full dynamics can be unstable, requires a lot of data, and can suffer from poor extrapolation (Kamp, Ultsch, and Brembeck 2023). As a result, hybrid modeling is an emerging field that combines the flexibility of neural networks with known physics and first principle models. Extensions like Universal Differential Equations (UDEs) (Rackauckas et al. 2020) or Physics-enhanced Neural ODEs (PeN-ODEs) (Kamp, Ultsch, and Brembeck 2023; Sorourifar et al. 2023) generalize this paradigm, allowing domain-specific knowledge to be incorporated into the model while still learning observable but unresolved effects. These approaches have demonstrated great success in various fields, including vehicle dynamics (Bruder and Mikelsons 2021; Thummerer, Stolar, and Mikelsons 2022), chemistry (Thebelt et al. 2022), climate modeling (Ramadhan et al. 2023), and process optimization (Misener and L. Biegler 2023).

Training neural components typically involves simulating (2) for some initial parameters  $\mathbf{p}$  and then propagating sensitivities of the ODE solver backward in each iteration. Afterward, the parameters are updated via gradient descent. This process is computationally expensive and results in long training times (Lehtimäki, Paunonen, and Linne 2024; Roesch, Rackauckas, and Stumpf 2021; Shapovalova and Tsay 2025), as explicit integrators are low-order and unstable, requiring small step sizes, while stable, implicit integrators involve solving nonlinear systems at each step, thus being computationally demanding.

To address these enormous training times several alternative procedures have been proposed: in (Roesch, Rackauckas, and Stumpf 2021) a collocation technique is introduced, which approximates the right hand side (RHS) of an ODE from data. The NN is then trained on the approximations with standard training frameworks. Further, in (Lehtimäki, Paunonen, and Linne 2024) model order reduction is used to accurately simulate the dynamics in low-dimensional subspaces. Very recent work presented in (Shapovalova and Tsay 2025) introduced global direct collocation with Chebyshev nodes, a method originating from dynamic optimization for optimal control and parameter optimization, for training Neural ODEs. The approach reduces the continuous training problem to a large finite dimensional nonlinear program (NLP) and shows fast and stable convergence, demonstrated on a typical problem, the Van-der-Pol oscillator.

In Modelica-based workflows, the training of Neural ODEs is typically performed externally by exporting a Functional Mock-Up Unit (FMU), subsequently training it in Python or Julia using standard machine learning frameworks and ODE solvers, and finally re-importing the hybrid model. While this approach introduces external dependencies and additional transformation steps, the *NeuralFMU* workflow (Thummerer, Stoljar, and Mikelsons 2022) demonstrates a practical method for integrating hybrid models into real-world applications.

Building on recent advances in direct collocation-based training of Neural ODEs (Shapovalova and Tsay 2025), we significantly extend the approach to PeN-ODEs. We formulate the training process as a dynamic optimization problem and discretize both neural and physical components using a stable, high-order implicit collocation scheme at flipped Legendre-Gauss-Radau (fLGR) points. This results in a large but structured NLP, allowing efficient, simultaneous optimization of states and parameters. Our custom, parallelized implementation leverages second-order information and the open-source NLP solver Ipopt (Wächter and L. T. Biegler 2006). It is designed for a future integration into the open-source modeling and simulation environment OpenModelica (Fritzson, Pop, Abdelhak, et al. 2020), thus providing an accessible training environment independent of external tools.

## 2 Dynamic Optimization for NODEs

In this section, we introduce a general class of dynamic optimization problems (DOPs) and formulate training for both NODEs and PeN-ODEs as instances of this class. We then discuss the transcription of the continuous problem into a large-scale nonlinear optimization problem (NLP). Finally, necessary considerations and key challenges are presented.

### 2.1 Generic Problem Formulation

Consider the DOP

$$\min_{\mathbf{p}} M(\mathbf{x}(t_0), \mathbf{x}(t_f), \mathbf{p}) + \int_{t_0}^{t_f} L(\mathbf{x}(t), \mathbf{p}, t) dt \quad (3a)$$

s.t.

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{p}, t) \quad \forall t \in T \quad (3b)$$

$$\mathbf{g}^L \leq \mathbf{g}(\mathbf{x}(t), \mathbf{p}, t) \leq \mathbf{g}^U \quad \forall t \in T \quad (3c)$$

$$\mathbf{r}^L \leq \mathbf{r}(\mathbf{x}(t_0), \mathbf{x}(t_f), \mathbf{p}) \leq \mathbf{r}^U \quad (3d)$$

for a fixed time horizon  $T = [t_0, t_f]$  with time variable  $t \in T$ . The states of the system are given by  $\mathbf{x} : T \rightarrow \mathbb{R}^{d_x}$  and the goal is to find optimal time-invariant parameters  $\mathbf{p} \in \mathbb{R}^{d_p}$ , such that the objective (3a) becomes minimal and the constraints (3b)–(3d) are satisfied. These constraints are divided into the ODE (3b) and *path constraints* (3c), which both must be satisfied at all times on time horizon  $T$ , as well as *boundary constraints* (3d), which must only hold at the initial and final time points  $t_0, t_f$ . The objective

is composed of a *Mayer* term  $M$ , that defines a cost at the boundary of  $T$ , and a *Lagrange* term  $L$ , that penalizes an accumulated cost over the entire time horizon. To ensure compatibility with typical nonlinear optimizers, all model functions must be twice continuously differentiable. This includes neural networks, their activation functions, and error measures. For completeness, the bounds of the constraints are given as  $\mathbf{g}^L, \mathbf{g}^U \in (\mathbb{R} \cup \{-\infty, \infty\})^{d_g}$  and  $\mathbf{r}^L, \mathbf{r}^U \in (\mathbb{R} \cup \{-\infty, \infty\})^{d_r}$ .

### 2.2 Reformulation of PeN-ODE Training

*PeN-ODEs* embed one or more NNs with parameters  $\mathbf{p}$  into known, possibly equation-based, dynamics  $\dot{\mathbf{x}}(t) = \hat{\boldsymbol{\phi}}(\mathbf{x}, \mathbf{u}, t)$ . The resulting differential equation has the form

$$\dot{\mathbf{x}}(t) = \boldsymbol{\phi}(\mathbf{x}, \mathbf{u}, t, NN_{\mathbf{p}}(\mathbf{x}, \mathbf{u}, t)), \quad (4)$$

where  $\mathbf{u} : T \rightarrow \mathbb{R}^{d_u}$  is a fixed input vector and  $NN_{\mathbf{p}}$  are enhancing NNs. This formalism aims to enhance systems that already express dynamics based on first principles, by further incorporating data-driven observables in form of neural components. Clearly, these components need not be NNs in general, and can be any parameter dependent expression. With additional information about the problem, one could use a polynomial, rational function, sum of radial basis functions or Fourier series.

The subsequent considerations also apply to the training of NODEs, where the goal is to learn the full dynamics without relying on a first principle model. This is evident from the fact that NODEs are a subclass of PeN-ODEs with

$$\dot{\mathbf{x}}(t) = NN_{\mathbf{p}}(\mathbf{x}, \mathbf{u}, t) = \boldsymbol{\phi}(\mathbf{x}, \mathbf{u}, t, NN_{\mathbf{p}}(\mathbf{x}, \mathbf{u}, t)). \quad (5)$$

In this paper, we propose a formulation for training PeN-ODEs as a DOP (3a)–(3d), using known data trajectories  $\hat{\mathbf{q}}$  and the corresponding predicted quantity  $\mathbf{q}$ . The DOP takes the form

$$\min_{\mathbf{p}} \int_{t_0}^{t_f} E(\mathbf{q}(\mathbf{x}, \mathbf{u}, t, NN_{\mathbf{p}}(\mathbf{x}, \mathbf{u}, t)), \hat{\mathbf{q}}(t)) dt \quad (6a)$$

s.t.

$$\dot{\mathbf{x}}(t) = \boldsymbol{\phi}(\mathbf{x}, \mathbf{u}, t, NN_{\mathbf{p}}(\mathbf{x}, \mathbf{u}, t)) \quad \forall t \in T \quad (6b)$$

for some smooth error measure  $E$ , e.g. the squared 2-norm  $E(\mathbf{q}, \hat{\mathbf{q}}) = \|\mathbf{q} - \hat{\mathbf{q}}\|_2^2$ . This formulation represents the minimal setup.

By further incorporating the generic constraints (3c) and (3d), it is possible to impose an initial or final condition on the states as well as enforce desired behavior. For example, consider a NN approximation of a force element  $NN_F$ , with no force acting in its resting position. Therefore, the NN should have a zero crossing, i.e.  $NN_F(0) = 0$ . This can be trivially formulated as a constraint, without introducing a penalty term that may distort the optimization as in standard unconstrained approaches like (Kamp,

Ultsch, and Brembeck 2023). Thus, the optimizer can handle the constraint appropriately.

As the loss in (6a) is a continuous-time integral, it enables an accurate and stable approximation using the same discretization employed for the system dynamics. In contrast to MSE loss as in (Shapovalova and Tsay 2025), which effectively corresponds to a first-order approximation of the integral, our formulation benefits from high-order quadrature, potentially preserving the accuracy of the underlying discretization.

### 2.3 Transcription with Direct Collocation

In the following, the general DOP (3a)–(3d) is reduced to a NLP using orthogonal direct collocation. Direct collocation approaches have proven to be highly efficient in solving DOPs and are implemented in a variety of free and commercial tools, such as PSOPT (Becerra 2010), CasADi (Andersson et al. 2019) or GPOPS-II (Patterson and A. V. Rao 2014), as well as in Modelica-based environments like OpenModelica (Ruge et al. 2014) or JModelica (Magnusson and Åkesson 2015). While OpenModelica only supports optimal control problems, the other frameworks allow for simultaneous optimization of static parameters. Recent work in the field of NODEs (Shapovalova and Tsay 2025) shows that learning the right hand side of small differential equations can be performed stably and efficiently using global collocation with Chebyshev nodes.

In direct collocation the states are approximated by piecewise polynomials, that satisfy the differential equation at so-called *collocation nodes*, usually chosen as roots of certain orthogonal polynomials. If the problem is smooth and with increasing number of collocation nodes, these methods achieve *spectral*, i.e. exponential, convergence to the exact solution. In this paper, the collocation nodes are chosen as the *flipped Legendre-Gauss-Radau points (fLGR)* rescaled from  $[-1, 1]$  to  $[0, 1]$ . This rescaling is performed, so that the corresponding collocation method is equivalent to the Radau IIA Runge-Kutta method. Radau IIA has excellent properties, since it is  $A$ -,  $B$ - and  $L$ -stable and achieves order  $2m - 1$  for  $m$  stages or collocation nodes. These nodes  $c_j$  for  $j = 1, \dots, m$  are given as the  $m$  roots of the polynomial  $(1 - t)P_{m-1}^{(1,0)}(2t - 1)$ , where  $P_{m-1}^{(1,0)}$  is the  $(m - 1)$ -th Jacobi polynomial with  $\alpha = 1$  and  $\beta = 0$ . A detailed explanation of the method's construction based on quadrature rules is given in (Langenkamp 2024).

First, we divide the time horizon  $[t_0, t_f]$  into  $n + 1$  intervals  $[t_i, t_{i+1}]$  for  $i = 0, \dots, n$  with length  $\Delta t_i := t_{i+1} - t_i$ . In each interval  $[t_i, t_{i+1}]$  the collocation nodes  $t_{ij} := t_i + c_j \Delta t_i$  for  $j = 1, \dots, m_i$  as well as the first grid point  $t_{i0} := t_i + c_0 \Delta t_i$  with  $c_0 = 0$  are added. Since the last node  $c_{m_i} = 1$  is contained in any Radau IIA scheme, the last grid point of interval  $i - 1$  exactly matches the first grid point of interval  $i$ , i.e.  $t_{i-1, m_{i-1}} = t_{i0}$ . Furthermore, the states are approximated as  $\mathbf{x}(t_{ij}) \approx \mathbf{x}_{ij}$  and for each interval  $i$  replaced by

a Lagrange interpolating polynomial  $\mathbf{x}_i(t) = \sum_{j=0}^{m_i} \mathbf{x}_{ij} l_j(t)$  of degree  $m_i$ , where

$$l_j(t) := \prod_{\substack{k=0 \\ k \neq j}}^{m_i} \frac{t - t_{ik}}{t_{ij} - t_{ik}} \quad \forall j = 0, \dots, m_i \quad (7)$$

are the Lagrange basis polynomials. Note that the parameters  $\mathbf{p}$  are time-invariant and thus, need not be discretized. Each  $\mathbf{x}_i$  must satisfy the differential equation (3b) at the collocation nodes  $t_{ij}$  and also match the initial condition  $\mathbf{x}_{i0}$ , which is given from the previous interval  $i - 1$ . By differentiating we get the collocated dynamics

$$\mathbf{0} = \begin{bmatrix} D_{10}^{(1)} I & \dots & D_{1m_i}^{(1)} I \\ \vdots & \ddots & \vdots \\ D_{m_i 0}^{(1)} I & \dots & D_{m_i m_i}^{(1)} I \end{bmatrix} \begin{bmatrix} \mathbf{x}_{i0} \\ \vdots \\ \mathbf{x}_{im_i} \end{bmatrix} - \Delta t_i \begin{bmatrix} \mathbf{f}_{i1} \\ \vdots \\ \mathbf{f}_{im_i} \end{bmatrix} \quad (8)$$

with identity matrix  $I \in \mathbb{R}^{d_x \times d_x}$ , entries of the first differentiation matrix  $D_{jk}^{(1)} := \frac{d\tilde{l}_k}{d\tau}(c_j)$ , where

$$\tilde{l}_k(\tau) := \prod_{\substack{r=0 \\ r \neq k}}^{m_i} \frac{\tau - c_r}{c_k - c_r} \quad \forall k = 0, \dots, m_i, \quad (9)$$

and the RHS of the ODE  $\mathbf{f}_{ij} := \mathbf{f}(\mathbf{x}_{ij}, \mathbf{p}, t_{ij})$ . The numerical values of  $D_{jk}^{(1)}$  can be calculated very efficiently with formulas provided in (Schneider and Werner 1986).

Approximating  $L$  is analogous to discretizing the differential equation. This is done by replacing the integral with a Radau quadrature rule of the form

$$\int_{t_0}^{t_f} L(\mathbf{x}(t), \mathbf{p}, t) dt \approx \sum_{i=0}^n \Delta t_i \sum_{j=1}^{m_i} b_j L(\mathbf{x}_{ij}, \mathbf{p}, t_{ij}), \quad (10)$$

where the quadrature weights are given by

$$b_j = \int_0^1 \prod_{\substack{k=1 \\ k \neq j}}^{m_i} \frac{\tau - c_k}{c_j - c_k} d\tau \quad \forall j = 1, \dots, m_i. \quad (11)$$

$M$  and the boundary constraints (3d) are approximated by replacing the values on the boundary with their discretized equivalents, i.e.  $M(\mathbf{x}(t_0), \mathbf{x}(t_f), \mathbf{p}) \approx M(\mathbf{x}_{00}, \mathbf{x}_{nm_n}, \mathbf{p})$  and  $\mathbf{r}(\mathbf{x}(t_0), \mathbf{x}(t_f), \mathbf{p}) \approx \mathbf{r}(\mathbf{x}_{00}, \mathbf{x}_{nm_n}, \mathbf{p})$ , while the path constraints (3c) are evaluated at all nodes, i.e.  $\mathbf{g}(\mathbf{x}(t_{ij}), \mathbf{p}, t_{ij}) \approx \mathbf{g}(\mathbf{x}_{ij}, \mathbf{p}, t_{ij})$ .

### 2.4 Training with Nonlinear Programming

By flattening the collocated dynamics (8), we obtain the discretized DOP (3a)–(3d) of the form

$$\min_{\mathbf{x}_{ij}, \mathbf{p}} M(\mathbf{x}_{00}, \mathbf{x}_{nm_n}, \mathbf{p}) + \sum_{i=0}^n \Delta t_i \sum_{j=1}^{m_i} b_j L(\mathbf{x}_{ij}, \mathbf{p}, t_{ij}) \quad (12a)$$

s.t.

$$\mathbf{0} = \sum_{k=0}^{m_i} D_{jk}^{(1)} \mathbf{x}_{ik} - \Delta t_i \mathbf{f}(\mathbf{x}_{ij}, \mathbf{p}, t_{ij}) \quad \forall i, \forall j \geq 1 \quad (12b)$$

$$\mathbf{g}^L \leq \mathbf{g}(\mathbf{x}_{ij}, \mathbf{p}, t_{ij}) \leq \mathbf{g}^U \quad \forall i, \forall j \geq 1 \quad (12c)$$

$$\mathbf{r}^L \leq \mathbf{r}(\mathbf{x}_{00}, \mathbf{x}_{nm_n}, \mathbf{p}) \leq \mathbf{r}^U \quad (12d)$$

This large-scale NLP (12a)–(12d) can be implemented and solved efficiently in nonlinear optimizers such as Ipopt (Wächter and L. T. Biegler 2006), SNOPT (P. E. Gill et al. 2007; Philip E. Gill, Murray, and Saunders 2005) or KNITRO (Byrd, Nocedal, and Waltz 2006). These NLP solvers exploit the sparsity of the problem as well as the first and second order derivatives of the constraint vector and objective function to converge quickly to a suitable local optimum.

The open-source interior-point method Ipopt requires the already mentioned first derivatives and, in addition, the Hessian of the augmented Lagrangian at every iteration. Since the derivatives only need to be evaluated at the collocation nodes, there is no need to propagate them as in traditional ODE solver-based training. Furthermore, training with ODE solvers usually limits itself to first order derivatives and therefore, does not utilize higher order information as in the proposed approach.

The resulting so-called *primal-dual* system is then solved using a linear solver for symmetric indefinite systems, such as the open-source solver MUMPS (Amestoy et al. 2001) or a proprietary solver from the HSL suite (HSL 2013), after which an optimization step is performed. This step updates all variables  $\mathbf{x}_{ij}, \mathbf{p}$  simultaneously, allowing for direct observation and adjustment of intermediate values. In contrast, ODE solver training only captures the final result after integrating the dynamics over time, without the ability to directly influence intermediate states during the optimization process. Because this linear system must be solved in every iteration anyway, high order, stable, implicit Runge-Kutta collocation methods, e.g. Radau IIA, can be embedded with only limited overhead. As a result, the NLP formulation overcomes key limitations of explicit ODE solvers in terms of order, stability, and allowable step size. Moreover, since the solver performs primal and dual updates, the solution does not need to remain feasible during the optimization, in contrast to ODE solver approaches where the dynamics (3b) are enforced at all times through forward simulation. This results in both advantages and disadvantages: On the one hand, it enables more flexible and aggressive updates, potentially accelerating convergence. On the other hand, it may lead to intermediate solutions that temporarily violate physical consistency or produce invalid function evaluations, which require careful handling.

For a comprehensive overview of nonlinear programming, interior-point methods, and their application to collocation-based dynamic optimization, we refer interested readers to (L. T. Biegler 2010) and the Ipopt implementation paper (Wächter and L. T. Biegler 2006).

## 2.5 Challenges and Practical Aspects

We identify four main challenges in training PeN-ODEs using the proposed direct collocation and nonlinear programming approach. These challenges are closely related to those encountered in conventional PeN-ODE or general NN training.

### 2.5.1 Grid Selection

The choice of time grid  $\{t_0, \dots, t_{n+1}\}$  and the number of collocation nodes per interval  $m_i$  are crucial for both the accuracy and efficiency of the training process. In practice, the grid can either be chosen equidistant or tailored to the specific problem. While equidistant grids are straightforward to implement and often sufficient for well-behaved systems, non-equidistant grids may reduce computational costs while capturing the dynamics more efficiently. Placing more intervals with low degree collocation polynomials in regions of rapid state change can improve approximation quality without unnecessarily increasing the problem size. Similarly, in well-behaved regions, it is feasible to perform larger steps with more collocation nodes. Because the collocation scheme and grid are embedded into the NLP, these must be given *a-priori*. This leaves room for future developments of adaptive mesh refinement methods with effective mesh size reduction, which have already shown great success for optimal control problems (Zhao and Shang 2018; Liu, Hager, and A. Rao 2015).

### 2.5.2 Initial Guesses

Due to the size and possible nonlinearity of the resulting NLP, the choice of initial guesses has a strong influence on convergence behavior. Unlike classical NN training, where poor initialization primarily affects convergence speed, the constrained nature of the transcribed dynamic optimization problem can lead to poor local optima or even solver failure. It is therefore of high importance to perform informed initializations for the states  $\mathbf{x}_{ij}$  and, if possible, for the NN parameters  $\mathbf{p}$ .

One practical approach to obtain the required parameter guesses is to first train the network on a small, representative subset of the full dataset using constant initial values for both the states and parameters. The optimized parameters resulting from this reduced problem then serve as informed initial guesses for the full training problem. Consequently, the states are obtained by simulation, i.e.  $\mathbf{x}(t) := \text{ODESolve}(\boldsymbol{\phi}(\mathbf{x}, \mathbf{u}, t, \mathbf{NN}_{\mathbf{p}}(\mathbf{x}, \mathbf{u}, t), \mathbf{x}_0))$  and  $\mathbf{x}_{ij} := \mathbf{x}(t_{ij})$  for a given initial condition  $\mathbf{x}(t_0) = \mathbf{x}_0$ . By construction, the collocated dynamic constraints (12b) are satisfied, leading to improved convergence and stability in the full NLP.

Clearly, this strategy does not work in general. However, in simple cases where the model can be decomposed and the NN’s input-output behavior is observable, e.g. if model components should be replaced by a neural surrogate, the NN can be pre-trained using standard gradient descent. This yields reasonable initial guesses



for the parameters and states by simulation, which can then be integrated into the constrained optimization problem. Another pre-training strategy could be the *collocation technique* proposed in (Roesch, Rackauckas, and Stumpf 2021). Still, developing general, effective strategies to obtain reasonable initial guesses is one of the key challenges and limiting factors we identify for the general application of this approach.

### 2.5.3 Batch-wise Training

Standard ML frameworks employ batch learning to efficiently split up data. This is not as straightforward when training with the approach described here. One might assume that the entire dataset must be included in a single discretized DOP. However, recent work (Shapovalova and Tsay 2025) demonstrates that batch-wise training is possible and promises significant potential. The Alternating Direction Method of Multipliers (ADMM) (Boyd et al. 2011) allows decomposing the optimization problem into smaller subproblems that can be trained independently, while enforcing consensus between them. This allows for memory-efficient training and opens up the possibility of handling larger models or learning from multiple data trajectories simultaneously.

### 2.5.4 Training of Larger Networks

While computing Hessians of NNs is generally expensive, it is tractable for small networks. To reduce computational effort for larger networks, it might be reasonable to use partial Quasi-Newton approximations such as SR1, BFGS or DFP (L. T. Biegler 2010) to approximate the dense parts of the augmented Lagrangian Hessian  $H$ , e.g. the blocks  $H_{x_{ij},p}$  and  $H_{pp}$ , or solely  $H_{pp}$ . These blocks, which contain derivatives with respect to the NN parameters, are computationally expensive, while the block  $H_{x_{ij},x_{ij}}$ , which contains second derivatives with respect to the collocated states, is extremely sparse and comparably cheap. A Quasi-Newton approximation of the block  $H_{x_{ij},x_{ij}}$  is therefore disadvantageous. The sparsity can be exploited by computing this block analytically and using it directly in the Quasi-Newton update. An implementation of this partial update using the SR1 Quasi-Newton method is straightforward. Instead of one expensive symmetric rank-one update for the entire Hessian  $H$ , one cheap symmetric rank-one update for  $H_{pp}$  and one general rank-one update for  $H_{x_{ij},p}$  are needed.

This procedure significantly reduces the cost of the Hessian, while still providing fairly detailed derivative information. SR1 is particularly advantageous here, as it can represent indefinite Hessians, which is favorable when dealing with highly nonlinear functions.

Since our current examples perform well with small NNs, we do not explore larger networks in this paper. However, we anticipate that such Quasi-Newton strategies will be necessary in future work with larger networks. Very recent work (Lueg et al. 2025) independently expresses similar ideas, highlighting the potential of the ap-

proach.

## 3 Implementation

The generic DOP (3a)–(3d) and its corresponding NLP formulation (12a)–(12d) are implemented in the custom open-source framework GDOPT (Langenkamp 2024), which is publicly available.<sup>1</sup> For neural network training the code has been extended, including predefined parametric blocks such as neural networks, support for data trajectories and parallelized optimizations. This extended, experimental version is also publicly available.<sup>2</sup>

### 3.1 GDOPT

GDOPT (General Dynamic Optimizer) consists of two main components: an expressive Python-based package *gdopt* and an efficient C++ library *libgdopt*. The Python interface provides an user-friendly modeling environment and performs symbolic differentiation and code generation. Symbolic expressions are optimized using common subexpression elimination via SymEngine<sup>3</sup>, and the resulting expressions together with first and second derivatives are translated into efficient C++ callback functions for runtime evaluation. In the present implementation, all expressions are flattened, resulting in large code, especially for the Hessian. Note that keeping NNs vectorized and employing symbolic differentiation rules with predefined NN functions offers significant advantages.

The library *libgdopt* implements a generalized version of the NLP (12a)–(12d) using Radau IIA collocation schemes, while also supporting optimal control problems. It is interfaced with Ipopt to solve the resulting nonlinear programs. Both symbolic Jacobian and Hessian rely on exact sparsity patterns discovered in the Python interface. Additional functionality includes support for nominal values, initial guesses, runtime parameters, mesh refinements, plotting utilities, and special functions. A detailed overview of features and modeling is provided in the GDOPT User’s Guide<sup>4</sup>.

Nevertheless, GDOPT lacks important capabilities that established modeling languages and tools offer, such as object-oriented, component-based modeling and support for differential-algebraic equation (DAE) systems. It is possible to model DAEs by introducing control variables for algebraic variables. However, this approach increases the workload and size of the NLP and may lead to instabilities.

### 3.2 Parallel Callback Evaluations

Since in every optimization step, the function evaluations as well as first and second derivatives of all NLP components (12a)–(12d) must be provided to Ipopt, an efficient callback evaluation is crucial to accelerate the training.

<sup>1</sup><https://github.com/linuslangenkamp/GDOPT>

<sup>2</sup>[https://github.com/linuslangenkamp/GDOPT\\_DEV](https://github.com/linuslangenkamp/GDOPT_DEV)

<sup>3</sup><https://github.com/symengine/symengine>

<sup>4</sup><https://github.com/linuslangenkamp/GDOPT/blob/master/usersguide/usersguide.pdf>



Note that these callbacks themselves consist of the continuous functions evaluated at all collocation nodes. For simplicity we write  $z_{ij} := [\mathbf{x}_{ij}, \mathbf{p}]^T$  for the variables at a given collocation node and, in addition,

$$\psi(\mathbf{x}, \mathbf{p}, t) := [L(\cdot), \mathbf{f}(\cdot), \mathbf{g}(\cdot)]^T \quad (13)$$

for the vector of functions that are evaluated at all nodes. Clearly, the required callbacks

$$\psi|_{z_{ij}} \quad \nabla \psi|_{z_{ij}} \quad \nabla^2 \psi|_{z_{ij}} \quad \forall i \forall j \geq 1$$

are independent with respect to the given collocation nodes  $t_{ij}$  and thus, allow for a straightforward parallelization. In the extension of GDOPT, we use OpenMP (Chandra et al. 2001) to parallelize the callback evaluations required by Ipopt. Depending on the specific callback, i.e. objective function, constraint violation, gradient, Jacobian, or Hessian of the augmented Lagrangian, a separate `omp parallel for` loop is used to evaluate the corresponding components at all collocation nodes. This design results in a significant reduction in computation time, especially for the comparably expensive dense derivatives of neural components.

## 4 Performance

In order to test the proposed training method and parallel implementation, two example problems are considered. The first example is the Quarter Vehicle Model (QVM) from (Kamp, Ultsch, and Brembeck 2023), where an equation-based model is enhanced with small neural components, such that physical behavior is represented more accurately. The second example is a standard NODE, where the dynamics of a Van-der-Pol oscillator (Roesch, Rackauckas, and Stumpf 2021) are learned purely from data. In both cases, the experimental setup closely follows the configurations used in the respective paper. Training is performed on a laptop running Ubuntu 24.04.2 with Intel Core i7-12800H (20 threads), 32 GB RAM and using GCC v13.3.0 with flags `-O3 -ffast-math` for compilation, while MUMPS (Amestoy et al. 2001) is used to solve linear systems arising from Ipopt (Wächter and L. T. Biegler 2006). All dependencies are free to use and open-source.

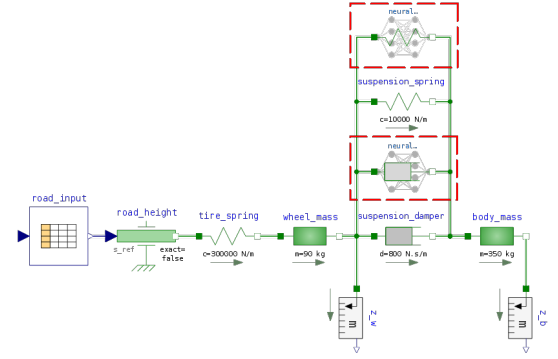
### 4.1 Quarter Vehicle Model

We follow the presentation in (Kamp, Ultsch, and Brembeck 2023) for an overview of the model and the data generation process. The Quarter Vehicle Model (QVM) captures the vertical dynamics of a road vehicle by modeling one wheel and the corresponding quarter of the vehicle body. It consists of two masses connected by spring-damper elements representing the suspension and tire dynamics. The linear base model is described by the differential equations  $\dot{z}_r = u$ ,  $\dot{z}_b = v_b$ ,  $\dot{z}_w = v_w$ , and

$$\dot{v}_b := a_b = m_b^{-1} (c_s \Delta z_s + d_s \Delta v_s) \quad (14a)$$

$$\dot{v}_w := a_w = m_w^{-1} (c_t \Delta z_t + d_t \Delta v_t - c_s \Delta z_s - d_s \Delta v_s) \quad (14b)$$

where  $\Delta z_s = z_w - z_b$ ,  $\Delta v_s = v_w - v_b$ ,  $\Delta z_t = z_r - z_w$ ,  $\Delta v_t = u - v_w$ . In addition,  $m_w$  is the mass of the wheel,  $m_b$  is mass of the quarter body, and  $c_s$  and  $d_s$  are the coefficients of the linear spring-damper pair between these masses, modeling the suspension. Furthermore,  $c_t$  and  $d_t$  define an additional linear spring-damper pair between the tire and ground. The state vector  $\mathbf{x} = [z_b, z_w, v_b, v_w, z_r]^T$  contains the positions of the body  $z_b$  and wheel  $z_w$ , their velocities  $v_b$  and  $v_w$ , and the road height  $z_r$ . The differential road height  $\dot{z}_r = u$  is given as an input and the observable outputs  $y = [a_w, a_b]$  measure the wheel and body accelerations. A corresponding Modelica model of the linear QVM is depicted in Figure 1. (Kamp, Ultsch, and Brembeck 2023)



**Figure 1.** Modelica Models of the Linear (without boxes) and Neural (with boxes) QVM. Modified from T. Kamp.

#### 4.1.1 Data Generation

The linear model is extended by introducing two additional nonlinear forces between both masses, a translational friction force  $F_{fr}$  and a progressive spring characteristic  $F_{pr}$ . These nonlinear components are introduced only for data generation, creating a more complicated, nonlinear model whose behavior deviates from the known base dynamics. Therefore, the differential equations (14a) and (14b) become

$$\dot{v}_b = m_b^{-1} (c_s \Delta z_s + d_s \Delta v_s + F_{pr}(\Delta z_s) + F_{fr}(\Delta v_s)) \quad (15a)$$

$$\dot{v}_w = m_w^{-1} (c_t \Delta z_t + d_t \Delta v_t - c_s \Delta z_s - d_s \Delta v_s - F_{pr}(\Delta z_s) - F_{fr}(\Delta v_s)) \quad (15b)$$

To generate data, a simulation of the nonlinear model for an imitation of a realistic, rough road (ISO8608, Type D (Můčka, Peter 2018)) is performed. The observables  $a_b$  and  $a_w$  as well as the states are disturbed by random Gaussian noise and sampled with 1000 Hz for a 42 s trajectory as in (Kamp, Ultsch, and Brembeck 2023).

#### 4.1.2 Training Setup

The nonlinear components  $F_{fr}(\Delta v_s)$  and  $F_{pr}(\Delta z_s)$  introduced for data generation are replaced by two neural network surrogates  $F_{fr}^{NN}(\Delta v_s)$  and  $F_{pr}^{NN}(\Delta z_s)$ , illustrated in Figure 1. The goal is to find suitable replacements that minimize the mismatch between the simulated, disturbed

outputs  $\hat{a}_b$  and  $\hat{a}_w$  of the nonlinear model and the observed data during the optimization. As discussed before, we write this objective as an integral over the entire time horizon, i.e.

$$\min \int_{t_0}^{t_f} \left( \frac{\hat{a}_b - a_b}{\sigma_{\hat{a}_b}} \right)^2 + \left( \frac{\hat{a}_w - a_w}{\sigma_{\hat{a}_w}} \right)^2 dt, \quad (16)$$

where  $\sigma_{\hat{a}_b}$  and  $\sigma_{\hat{a}_w}$  are corresponding standard deviations of the data, ensuring that both accelerations contribute equally to the objective. Furthermore, it is known that both force elements have a zero crossing and therefore, the additional constraints

$$F_{fr}^{NN}(0) = 0 \quad \text{and} \quad F_{pr}^{NN}(0) = 0 \quad (17)$$

are simply added to the optimization problem.

We employ three different training strategies. At first, both feedforward neural networks are trained directly on the full trajectory using randomly initialized parameters, while the initial state guesses are obtained from a simulation of the linear model (I). Each network has the structure  $1 \times 5 \rightarrow 5 \times 5 \rightarrow 5 \times 1$  and therefore, both nets contain just 92 parameters in total. We use the smooth squareplus activation function

$$\text{squareplus}(x) := \frac{x + \sqrt{1 + x^2}}{2} \quad (18)$$

to ensure a twice continuously differentiable NN as required for Ipopt. In the second strategy (II), described in Section 2.5.2, first an acceleration scheme is employed, where the same networks are trained on a short segment one eighth of the entire trajectory. After that, a simulation of the neural QVM with the obtained parameters is performed. The resulting states are used as initial guesses in the subsequent optimization with full data. To show that the surrogates need not be NNs and training can be performed efficiently with other parameter-dependent expressions, the third strategy (III) uses rational functions to model the unknown behavior. For instance,  $F_{fr}$  is replaced by

$$F_{fr}^{RC}(\Delta v_s) := \frac{\sum_{k=0}^N \omega_k T_k(\Delta v_s)}{\sum_{k=0}^D \theta_k T_k(\Delta v_s)}, \quad (19)$$

where  $T_k$  is the  $k$ -th Chebyshev polynomial,  $\omega_k$  and  $\theta_k$  are parameters to be optimized, and  $N$  and  $D$  are the numerator and denominator degrees. For both rational functions we choose  $N = D = 7$ , resulting in a total of merely 32 learnable parameters.

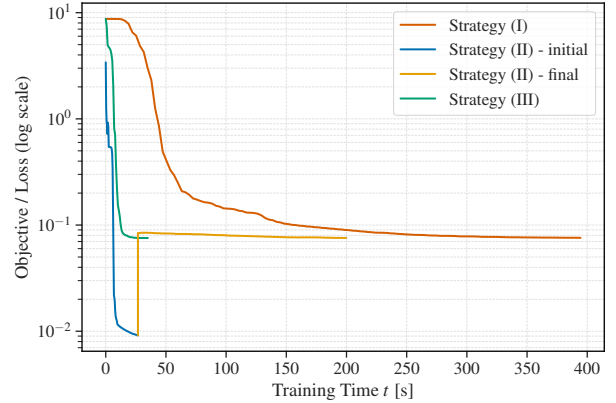
Since the QVM contains very fast dynamics due to high-frequency excitations, in all cases the time horizon is divided into a tightly spaced, equidistant grid of 2500 intervals and using a constant 5-step Radau IIA collocation scheme of order 9. This leads to a total of 12500 collocation nodes and more than  $2.7 \times 10^6$  nonzeros in the Jacobian and roughly  $4.73 \times 10^6$  nonzeros in the Hessian of the large-scale NLP.

#### 4.1.3 Results

Table 1 presents the training times for all strategies. Each optimization was run for a maximum of 150 NLP iterations and was automatically terminated early if no further significant improvement in objective could be achieved. We want to stress that all trainings, performed on a laptop, are executed in under 7 minutes, compared to 4.5 hours for the fastest optimization in (Kamp, Ultsch, and Brembeck 2023) using ODE solver-based training. Clearly, this is also due to the fact that smaller neural components are used. Furthermore, Figure 2 depicts the objective value with respect to training time, where both the first and second optimizations of (II) are concatenated.

Strategy	Total	Ipopt	Callbacks
(I)	394.43	284.18	110.25
(II) - initial	26.75	15.08	11.67
(II) - final	173.06	124.65	48.41
(II)	199.81	139.73	60.07
(III)	34.96	27.10	7.85

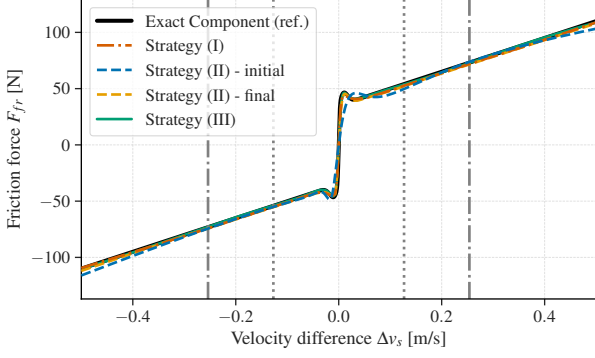
**Table 1.** Training Times in Seconds



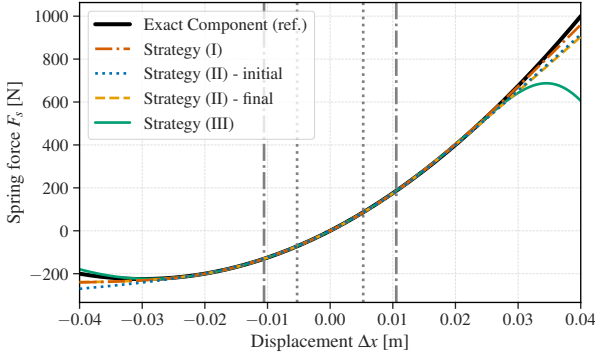
**Figure 2.** Objective History with Respect to Training Time

Even though a poor initialization has been used and thus the optimization required more time, the naive strategy (I) converged stably to a suitable optimum with a similar objective as strategies (II) and (III). For this example, the acceleration scheme (II) proves effective, since the initial optimization for a shorter data trajectory takes just 26.75 s in total, and the subsequent initial guesses for states and parameters become very good approximations of the real solution. This can be observed in Figure 3 and Figure 4, where the resulting neural surrogates are depicted. By performing the second optimization, (II) effectively halves the time required by strategy (I), i.e. less than 3.5 minutes, and furthermore results in indistinguishable neural components.

Moreover, as seen in Figure 3 and Figure 4, the resulting very small NNs match the reference in almost perfect



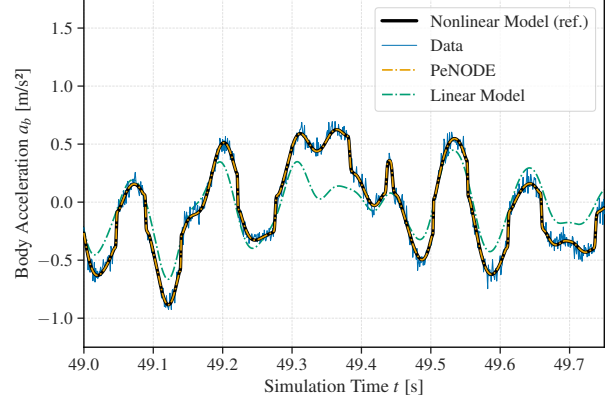
**Figure 3.** Neural and Reference Damper Characteristics



**Figure 4.** Neural and Reference Spring Characteristics

accordance, correctly representing highly nonlinear parts of the damper. These also generalize in a very natural way, as can be seen from the behavior outside the vertical dotted lines, which represent the first and second standard deviations of the inputs to the nets. Performing simulations on a new, unknown input road shows that the obtained PeNODE from (II) matches the nonlinear reference model perfectly, as illustrated in Figure 5. Note that, our characteristics of the damper and spring serve as even better surrogates than those reported in (Kamp, Ultsch, and Brembeck 2023), despite using significantly smaller networks and requiring considerably shorter training times. While (Kamp, Ultsch, and Brembeck 2023) relied on larger models with longer ODE solver-based training, our approach yields more accurate and better-generalizing results.

By having principal knowledge of the underlying characteristics shown in Figure 3 and Figure 4, it is possible to model observed behavior with minimal parameters. Since the number of Hessian nonzeros grows quadratically with the number of parameters, such knowledge of the procedure can greatly benefit both training time and surrogate quality. Therefore, consider simple rational functions as an educated guess for both missing components. This optimization, with 32 instead of 92 free parameters, is performed in under 35 seconds without any acceleration strategy. Moreover, the obtained surrogates are of mostly equal quality to the NN components from (II), although



**Figure 5.** Body Accelerations  $a_b$  for Simulations of PeNODE and Standard Models on a Type C Road (Múčka, Peter 2018)

Figure 4 shows that the rational function does not generalize as well. Nevertheless, these results demonstrate that unknown behavior may be expressed with fewer parameters and yield equivalent quality.

#### 4.1.4 Parallel Implementation

Finally, it is stressed that the parallel implementation, described in Section 3.2, leads to 5.44 times less time taken in the generated function callbacks. This yields a total training time that is more than halved and clearly shows that the implementation efficiently exploits the independence of collocation nodes.

Method	Total	Ipopt	Callbacks
GDOPT (default)	385.90	122.52	263.38
GDOPT (parallel)	173.06	124.65	48.41

**Table 2.** Comparison of Parallel and Sequential Optimization Times in Seconds of (II) - final

## 4.2 Van-der-Pol Oscillator

To illustrate the ability of learning a full NODE, we follow an example from (Roesch, Rackauckas, and Stumpf 2021), where a different kind of collocation method was proposed. This method approximates the RHS of the ODE with data, thus enabling faster, unconstrained training without the need for ODE solvers. Consider the Van-der-Pol (VdP) oscillator

$$\dot{x} = y \quad (20a)$$

$$\dot{y} = \mu y (1 - x^2) - x \quad (20b)$$

with  $\mu = 1$  and initial conditions  $x(t_0) = 2$  and  $y(t_0) = 0$ . Data generation is performed by simulating the dynamics with OpenModelica (Fritzson, Pop, Abdelhak, et al. 2020) on an equidistant grid with 200 intervals and artificially perturbing the observed states by additive Gaussian noise. To test sensitivities of the approach, we use 3 different

levels of noise  $\mathcal{N}(0, \sigma)$  to disturb the observable states, i.e. no noise ( $\sigma = 0$ ), low noise ( $\sigma = 0.1$ ) and high noise ( $\sigma = 0.5$ ).

The continuous DOP has the form

$$\min_{\mathbf{p}_x, \mathbf{p}_y} \int_{t_0}^{t_f} (\hat{x}_\sigma - x)^2 + (\hat{y}_\sigma - y)^2 dt + \lambda \|\mathbf{p}\|_2^2 \quad (21a)$$

s.t.

$$\dot{x} = NN_{\mathbf{p}_x}^x(x, y) \quad (21b)$$

$$\dot{y} = NN_{\mathbf{p}_y}^y(x, y) \quad (21c)$$

$$x(t_0) = 2, y(t_0) = 0, \quad (21d)$$

where  $\hat{x}_\sigma, \hat{y}_\sigma$  is the disturbed state data,  $NN_{\mathbf{p}_x}^x(x, y)$ ,  $NN_{\mathbf{p}_y}^y(x, y)$  are neural networks of the architecture  $2 \times 5 \rightarrow 5 \times 5 \rightarrow 5 \times 1$  with sigmoid activation function,  $\mathbf{p} = [\mathbf{p}_x, \mathbf{p}_y]^T$  are 102 learnable parameters, while  $\lambda > 0$  is a regularization factor to enhance stability.

As in (Roesch, Rackauckas, and Stumpf 2021), the training is performed on a 7 second time horizon, thus including a little over one period of the oscillator. Furthermore, 500 intervals and the 5-step Radau IIA method of order 9 are used. The initial guesses for the state variables are trivially chosen as the constant initial condition and the NN parameters are initialized randomly. No acceleration strategy or simulation is used for educated initial guesses. We set a maximum number of Ipopt iterations / epochs of 200 and an optimality tolerance of  $10^{-7}$ .

#### 4.2.1 Results

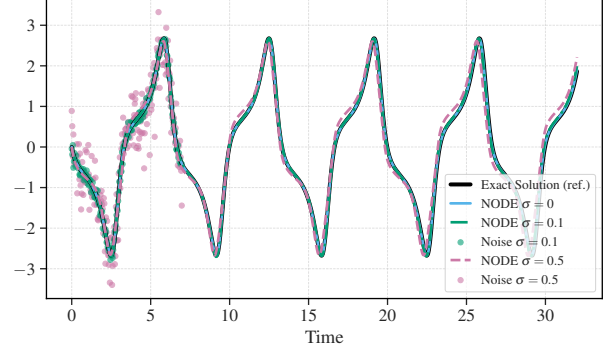
In almost all cases, the optimization terminates prematurely, since the optimality tolerance is fulfilled and thus, a local optimum was found. The corresponding training times are displayed in Table 3. Note that, because the high noise ( $\sigma = 0.5$ ) leads to larger objective values, the regularization  $\lambda$  is increased in this case. Nevertheless, the optimization with a total of 2500 discrete nodes is very rapid and terminates in several seconds from an extremely poor initial guess, while in some instances runs settle in poor local optima. Clearly, more reasonable initial guesses and sophisticated initialization strategies will further enhance these times and stability.

$\sigma$	$\lambda$	Total	Ipopt	Callbacks	#Epochs
0	$10^{-4}$	8.17	5.95	2.22	80
0.1	$10^{-4}$	7.69	5.60	2.09	76
0.5	$10^{-3}$	13.49	9.79	3.70	134

**Table 3.** Training Times (in seconds) and Number of Ipopt Iterations / Epochs of the Van-der-Pol Oscillator NNs

Figure 6 presents the simulation results and training data for various levels of noise. It is evident that for no or low noise, the solution obtained is virtually indistinguishable from the reference, which is quite impressive. Even with high noise ( $\sigma = 0.5$ ), the Neural ODE remains close

to the true solution, showing significantly better performance compared to the results reported in (Roesch, Rackauckas, and Stumpf 2021), where NODE and reference do not align for  $\sigma \geq 0.2$ . While smaller neural networks are employed here, these compact models still demonstrate exemplary performance and fast training, highlighting the effectiveness of the approach under severe noise.

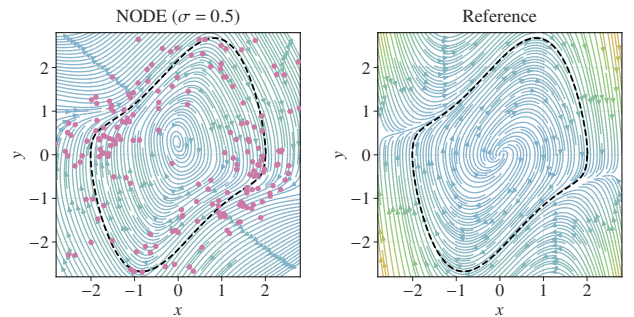


**Figure 6.** Simulation Results of the Neural and Reference Models as well as the Data for  $y(t)$

To verify the robustness of the training procedure, a comprehensive sensitivity analysis, consisting of 100 training runs for each noise level, was conducted. The results, detailed in the Appendix and Figure 10, demonstrate that while all runs converge perfectly for the no-noise case ( $\sigma = 0$ ) and approximately 95% show excellent agreement under low noise ( $\sigma = 0.1$ ), as expected the robustness diminishes with high noise ( $\sigma = 0.5$ ). In these cases, several runs converge to poor local optima or fail to converge, leading to solutions with noticeable period and amplitude mismatches or an outright collapse of the trajectory.

To further illustrate the obtained NODEs, we compare the learned and true vector fields of the ODE

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} \mapsto \begin{bmatrix} NN_{\mathbf{p}_x}^x(x, y) \\ NN_{\mathbf{p}_y}^y(x, y) \end{bmatrix}, \quad \begin{bmatrix} x \\ y \end{bmatrix} \mapsto \begin{bmatrix} y \\ \mu y (1 - x^2) - x \end{bmatrix}. \quad (22)$$



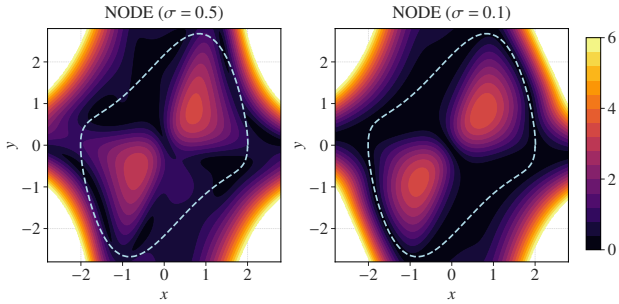
**Figure 7.** Neural ( $\sigma = 0.5$ ) and Reference Vector Fields of the ODE and the Exact VdP Trajectory (dashed)

In Figure 7, we show the high-noise NODE, its training data, and the true vector field. Despite the heavily



scattered observable states, the NODE still manages to recover a vector field that aligns well with the true dynamics in the vicinity of the solution trajectory.

Further comparison is given in Figure 8, where the scalar fields visualize the 2-norm error between the neural and reference vector fields. The low-noise NODE ( $\sigma = 0.1$ ) yields way smaller error values along the trajectory, but even the high-noise model produces a fairly accurate vector field in regions close to available training data. As expected, generalization outside this domain remains limited, but within the training region, the results demonstrate strong consistency.



**Figure 8.** Scalar Fields Representing the 2-Norm Error Between the Neural ( $\sigma = 0.5$ ,  $\sigma = 0.1$ ) and Reference Vector Fields (Values  $> 6$  are white)

In addition to the local collocation approach with 500 intervals, we also reproduce the results in (Shapovalova and Tsay 2025) using a global, *spectral* collocation method. We employ a single interval with 70 fLGR nodes, corresponding to Radau IIA of order 139. This high-order global formulation yields an approximation of equal quality to the figures above, even under high noise ( $\sigma = 0.5$ ). Furthermore, the optimization terminates in outstandingly fast time, i.e. under 1.2 seconds. This demonstrates the remarkable efficiency and accuracy of spectral methods for such smooth problems.

## 5 Future Work

Although OpenModelica currently includes an optimization runtime implementing direct collocation (Ruge et al. 2014), it remains limited to basic features: it supports only 1- or 3-step Radau IIA collocation, does not allow parameter optimization, lacks analytic Hessians, and does not perform parallel callbacks. To overcome these limitations, work is underway to embed an extended version of *libgdopt* into OpenModelica. This integration combines recent developments from GDOPT with the existing strengths of OpenModelica, particularly its native ability to handle DAEs. The extended framework enables expressive Modelica-based modeling, native support of neural components via the NeuralNetwork Modelica library and incorporates recent advancements in mesh refinement for optimal control problems (Langenkamp 2024).

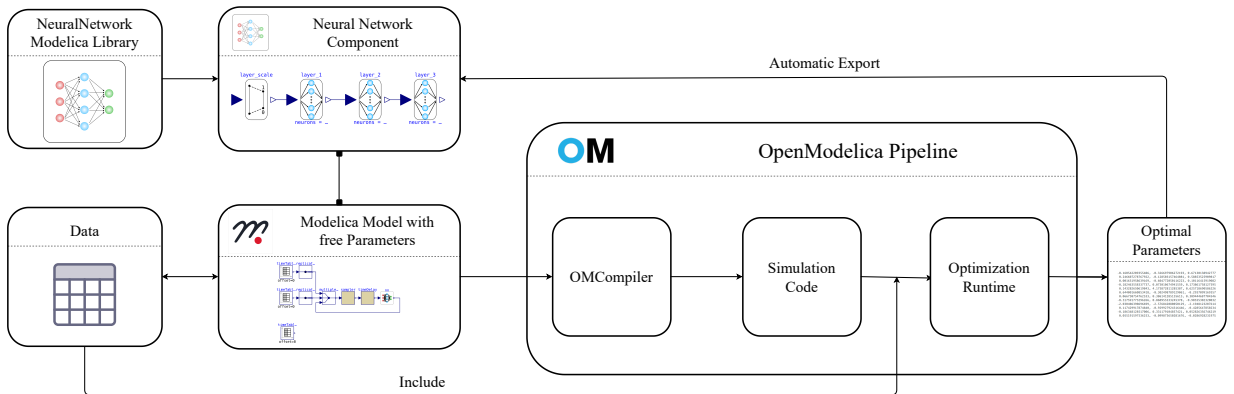
### 5.1 NeuralNetwork Modelica Library

The NeuralNetwork library, originally developed in (Codecà and Casella 2006), is an open-source Modelica library<sup>5</sup> modeling ML architectures with pure Modelica. Neural components can be constructed by connecting dense feedforward layers of arbitrary size with layers for PCA, standardizing, or scaling. These blocks contain all equations and parameters as pure Modelica code, which makes seamless integration of neural components into existing Modelica models straightforward. Together with this library, OpenModelica will enable modeling and training of PeN-ODEs within a single development environment.

### 5.2 Workflow

The workflow presented in Figure 9 illustrates the intended process for native Modelica-based modeling and training of PeN-ODEs. While some components of the workflow are operational, the full integration is still under active development. Users model physical systems and NN components with free parameters directly in Modelica and provide corresponding data. Both the model

<sup>5</sup><https://github.com/AMIT-HSBI/NeuralNetwork>



**Figure 9.** OpenModelica Workflow for PeN-ODE Training (under development)

and data are processed by the standard OpenModelica pipeline, while the OpenModelica Compiler (OMC) generates fast C code. The new backend of OMC introduces efficient array-size-independent symbolic manipulation (Abdelhak, Casella, and Bachmann 2023). Ongoing work further targets resizability of components after compilation (Abdelhak and Bachmann 2025), offering significant benefits for array-based components such as NNs.

Furthermore, current work focuses on leveraging the generated simulation code to enable fast callback functions for a new optimization runtime. After training, the optimal parameters are directly inserted into a NN block, enabling immediate simulations by swapping connectors.

This workflow removes the need for export and import steps for neural components and models, e.g. in the form of a Functional Mock-up Unit (FMU). It also eliminates reliance on external training routines in Julia or Python and avoids external C functions in the model, since the neural components are pure Modelica blocks. This integrated workflow unifies modeling, training, and simulation within a single toolchain, enabling free and accessible optimizations.

### 5.3 Neural DAEs

One of the main advantages of the integration into OpenModelica is the native ability to handle DAEs. Modelica compilers such as OpenModelica systematically apply index reduction and block-lower triangular (BLT) transformations to restructure DAEs into semi-explicit ODE form with index 1 (Ruge et al. 2014; Åkesson et al. 2012). As the simulation code already resolves algebraic variables during evaluations, no additional handling, e.g. inclusion of algebraic variables in the NLP, is needed on the optimization side. This allows the new training workflow to seamlessly extend from ODEs to DAEs, far surpassing the current range of applications.

## 6 Conclusion

This paper proposes a formulation of PeN-ODE training as a collocation-based NLP, simultaneously optimizing states and NN parameters. The approach overcomes key limitations of ODE solver-based training in terms of order, stability, accuracy, and allowable step size. The NLP uses high order quadrature for the NN loss, potentially preserving the accuracy of the discretization. We demonstrate that known physical behavior can be trivially enforced.

We provide an open-source parallelized extension to GDOPT and on two example problems demonstrate exemplary accuracy, training times, and generalization with smaller NNs compared to other training techniques, even under significant noise. Furthermore, we show that the approach allows for efficient optimization of other parameter dependent surrogates.

Key limitations and challenges of the proposed method, including grid selection and general initialization strategies to increase stability, as well as training with larger datasets and networks are identified. Addressing and eval-

uating these issues in future work is essential to support broader applicability. To enable accessible training of Neural DAEs, without relying on external tools, work is underway to implement the method in OpenModelica.

## Acknowledgements

This work was conducted as part of the OpenSCALING project (Grant No. 01IS23062E) at the University of Applied Sciences and Arts Bielefeld, in collaboration with Linköping University. The authors would like to express their sincere appreciation to both the OpenSCALING project and the Open Source Modelica Consortium (OSMC) for their support, collaboration, and shared commitment to advancing open-source modeling and simulation technologies.

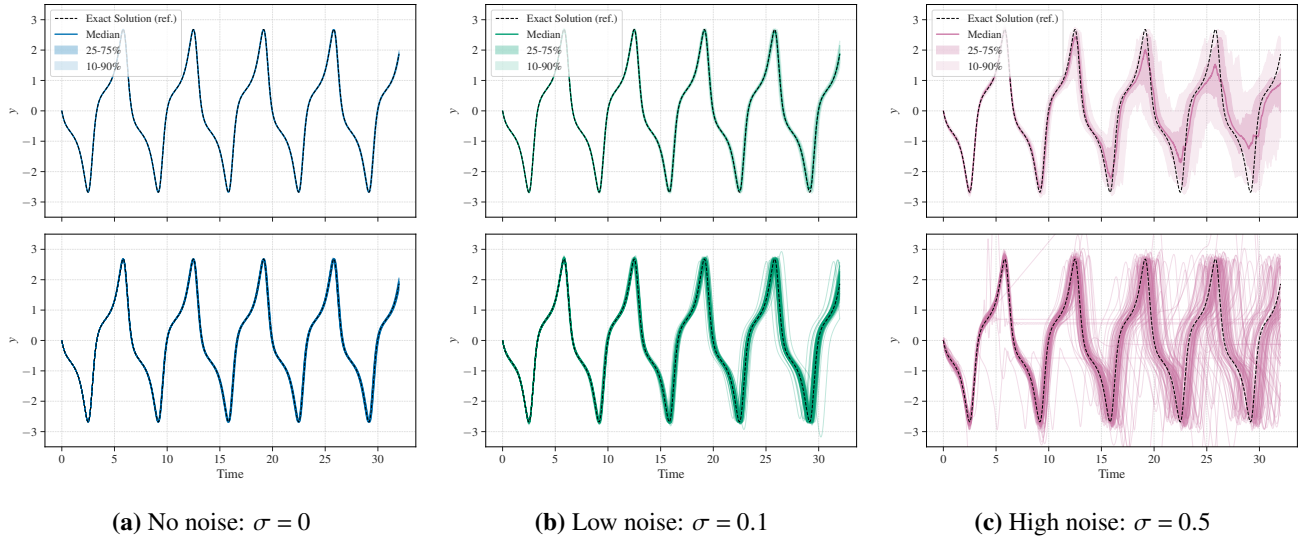
## References

- Abdelhak, Karim and Bernhard Bachmann (2025). *Compiler Status and Development of the New Backend*. Presentation at the 17th OpenModelica Annual Workshop - February 3, 2025, accessed on 2025-04-25. URL: <https://openmodelica.org/events/openmodelica-workshop/2025>.
- Abdelhak, Karim, Francesco Casella, and Bernhard Bachmann (2023-12). “Pseudo Array Causalization”. In: pp. 177–188. DOI: 10.3384/ecp204177.
- Åkesson, Johan et al. (2012). “Generation of Sparse Jacobians for the Function Mock-Up Interface 2.0”. In: *Book of abstracts / 9th International Modelica Conferenc*. Vol. 76. Linköping Electronic Conference Proceedings. Linköping University Electronic Press, pp. 185–196. DOI: 10.3384/ecp12076185.
- Amestoy, Patrick R. et al. (2001). “A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling”. In: *SIAM Journal on Matrix Analysis and Applications* 23.1, pp. 15–41.
- Andersson, Joel A E et al. (2019). “CasADi – A software framework for nonlinear optimization and optimal control”. In: *Mathematical Programming Computation* 11.1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.
- Becerra, V. M. (2010). “Solving complex optimal control problems at no cost with PSOPT”. In: *2010 IEEE International Symposium on Computer-Aided Control System Design*, pp. 1391–1396. DOI: 10.1109/CACSD.2010.5612676.
- Biegler, Lorenz T. (2010-01). *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. ISBN: 9780898717020. DOI: 10.1137/1.9780898719383.
- Boyd, Stephen et al. (2011). “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers”. In: *Foundations and Trends® in Machine Learning* 3.1, pp. 1–122. ISSN: 1935-8237. DOI: 10.1561/22000000016. URL: <http://dx.doi.org/10.1561/22000000016>.
- Bruder, Frederic and Lars Mikelsons (2021-09). “Modia and Julia for Grey Box Modeling”. In: pp. 87–95. DOI: 10.3384/ecp2118187.
- Byrd, Richard H., Jorge Nocedal, and Richard A. Waltz (2006). “Knitro: An Integrated Package for Nonlinear Optimization”. In: *Large-Scale Nonlinear Optimization*. Ed. by G. Di Pillo and M. Roma. Boston, MA: Springer US, pp. 35–59. ISBN: 978-0-387-30065-8. DOI: 10.1007/0-387-30065-1\_4. URL: [https://doi.org/10.1007/0-387-30065-1\\_4](https://doi.org/10.1007/0-387-30065-1_4).

- Chandra, Rohit et al. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann.
- Chen, Tian Qi et al. (2018). “Neural Ordinary Differential Equations”. In: *CoRR* abs/1806.07366. arXiv: 1806.07366. URL: <http://arxiv.org/abs/1806.07366>.
- Codecà, Fabio and Francesco Casella (2006-09). “Neural Network Library in Modelica”. In: *Proceedings of the 5th International Modelica Conference*. Vol. 2. Modelica Association. Vienna, Austria, pp. 549–557.
- Fritzson, Peter, Adrian Pop, Karim Abdelhak, et al. (2020-10). “The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development”. In: *Modeling, Identification and Control: A Norwegian Research Bulletin* 41, pp. 241–295. DOI: 10.4173/mic.2020.4.1.
- Gill, P. E. et al. (2007). *SNOPT 7.7 User's Manual*. Tech. rep. CCoM Technical Report 18-1. San Diego, CA: Center for Computational Mathematics, University of California, San Diego.
- Gill, Philip E., Walter Murray, and Michael A. Saunders (2005-01). “SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization”. In: *SIAM Rev.* 47.1, pp. 99–131. ISSN: 0036-1445. DOI: 10.1137/S0036144504446096. URL: <https://doi.org/10.1137/S0036144504446096>.
- HSL (2013). *HSL: A collection of Fortran codes for large-scale scientific computation*. Available at <http://www.hsl.rl.ac.uk>. Accessed: 15-04-2025.
- Kamp, Tobias, Johannes Ultsch, and Jonathan Brembeck (2023-11). “Closing the Sim-to-Real Gap with Physics-Enhanced Neural ODEs”. In: *20th International Conference on Informatics in Control, Automation and Robotics, ICINCO 2023*. Ed. by Guiseppina Gini, Henk Nijmeijer, and Dimitar Filev. Vol. 2. Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics. SCITEPRESS, pp. 77–84. URL: <https://elib.dlr.de/200100/>.
- Langenkamp, Linus (2024-12). *Adaptively Refined Mesh for Collocation-Based Dynamic Optimization*. Master's thesis. DOI: 10.13140/RG.2.2.18499.72484.
- Lehtimäki, Mikko, Lassi Paunonen, and Marja-Leena Linne (2024-01). “Accelerating Neural ODEs Using Model Order Reduction”. In: *IEEE Transactions on Neural Networks and Learning Systems* 35.1, pp. 519–531. ISSN: 2162-2388. DOI: 10.1109/tnnls.2022.3175757. URL: <http://dx.doi.org/10.1109/TNNLS.2022.3175757>.
- Liu, Fengjin, William Hager, and Anil Rao (2015-05). “Adaptive Mesh Refinement Method for Optimal Control Using Nonsmoothness Detection and Mesh Size Reduction”. In: *Journal of the Franklin Institute* 47. DOI: 10.1016/j.jfranklin.2015.05.028.
- Lueg, Laurens R. et al. (2025). *A Simultaneous Approach for Training Neural Differential-Algebraic Systems of Equations*. arXiv: 2504.04665 [cs.LG]. URL: <https://arxiv.org/abs/2504.04665>.
- Magnusson, Fredrik and Johan Åkesson (2015). “Dynamic Optimization in JModelica.org”. In: *Processes* 3.2, pp. 471–496. ISSN: 2227-9717. DOI: 10.3390/pr3020471. URL: <https://www.mdpi.com/2227-9717/3/2/471>.
- Misener, Ruth and Lorenz Biegler (2023). “Formulating data-driven surrogate models for process optimization”. In: *Computers & Chemical Engineering* 179, p. 108411. ISSN: 0098-1354. DOI: <https://doi.org/10.1016/j.compchemeng.2023.108411>. URL: <https://www.sciencedirect.com/science/article/pii/S0098135423002818>.
- Můčka, Peter (2018-01). “Simulated Road Profiles According to ISO 8608 in Vibration Analysis”. In: *Journal of Testing and Evaluation* 46, p. 20160265. DOI: 10.1520/JTE20160265.
- Patterson, Michael A. and Anil V. Rao (2014-10). “GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming”. In: *ACM Trans. Math. Softw.* 41.1. ISSN: 0098-3500. DOI: 10.1145/2558904. URL: <https://doi.org/10.1145/2558904>.
- Rackauckas, Chris et al. (2020-01). *Universal Differential Equations for Scientific Machine Learning*. DOI: 10.21203/rs.3.rs-55125/v1.
- Ramadhan, Ali et al. (2023). *Capturing missing physics in climate model parameterizations using neural differential equations*. arXiv: 2010.12559 [physics.ao-ph]. URL: <https://arxiv.org/abs/2010.12559>.
- Roesch, Elisabeth, Chris Rackauckas, and Michael Stumpf (2021-07). “Collocation based training of neural ordinary differential equations”. In: *Statistical Applications in Genetics and Molecular Biology* 20. DOI: 10.1515/sagmb-2020-0025.
- Ruge, Vitalij et al. (2014). “Efficient Implementation of Collocation Methods for Optimization using OpenModelica and ADOL-C”. In: *Proceedings of the 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*. Vol. 96. Linköping Electronic Conference Proceedings. Linköping University Electronic Press, pp. 1017–1025. DOI: 10.3384/ecp140961017.
- Schneider, C. and W. Werner (1986). “Some New Aspects of Rational Interpolation”. In: *Math. Comp.* 47.175, pp. 285–299. DOI: 10.1090/S0025-5718-1986-0842136-8. URL: <https://www.ams.org/journals/mcom/1986-47-175/S0025-5718-1986-0842136-8/>.
- Shapovalova, Mariia and Calvin Tsay (2025). *Training Neural ODEs Using Fully Discretized Simultaneous Optimization*. arXiv: 2502.15642 [cs.LG]. URL: <https://arxiv.org/abs/2502.15642>.
- Sorourifar, Farshud et al. (2023-09). “Physics-Enhanced Neural Ordinary Differential Equations: Application to Industrial Chemical Reaction Systems”. In: *Industrial & Engineering Chemistry Research* 62. DOI: 10.1021/acs.iecr.3c01471.
- Thebelt, Alexander et al. (2022-02). “Maximizing information from chemical engineering data sets: Applications to machine learning”. In: *Chemical Engineering Science* 252, p. 117469. DOI: 10.1016/j.ces.2022.117469.
- Thummerer, Tobias, Johannes Stoljar, and Lars Mikelsons (2022). “NeuralFMU: Presenting a Workflow for Integrating Hybrid NeuralODEs into Real-World Applications”. In: *Electronics* 11.19. ISSN: 2079-9292. DOI: 10.3390/electronics11193202. URL: <https://www.mdpi.com/2079-9292/11/19/3202>.
- Wächter, Andreas and Lorenz T. Biegler (2006-03). “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming”. In: *Mathematical Programming* 106.1, pp. 25–57. ISSN: 1436-4646. DOI: 10.1007/s10107-004-0559-y. URL: <https://doi.org/10.1007/s10107-004-0559-y>.
- Zhao, Jisong and Teng Shang (2018-09). “Dynamic Optimization Using Local Collocation Methods and Improved Multiresolution Technique”. In: *Applied Sciences* 8, p. 1680. DOI: 10.3390/app8091680.



## Appendix: Sensitivity Analysis and Robustness of VdP Neural ODEs



**Figure 10.** Sensitivity analysis of the learned Van-der-Pol (VdP) Neural ODEs to different levels of Gaussian noise. For each noise level, 100 training sessions were performed with random initial guesses for the neural network parameters and a random seed for the data perturbation. The plots show the simulation results for the observable state  $y(t)$  over an extended time horizon of 32 seconds. Each subplot consists of two panels: The top panel visualizes the reference solution (dashed black line), the median of the 100 learned solutions, and the 25-75% and 10-90% percentile bands. The bottom panel shows all 100 individual learned trajectories (faint lines). **(a) No noise ( $\sigma = 0$ ):** The method demonstrates full robustness and perfect convergence, with all 100 solutions converging to the exact reference solution. **(b) Low noise ( $\sigma = 0.1$ ):** The method remains highly robust. A vast majority of the solutions (approximately 95%) form a tight band around the reference, showing excellent agreement. **(c) High noise ( $\sigma = 0.5$ ):** As expected under severe noise, the robustness is reduced. While many solutions remain relatively close to the reference, some diverge significantly, indicating a failure to find a good local optimum during training. The solutions that do converge often show a period mismatch or diverge after a few periods, but still capture the general oscillatory behavior. The combined effects of reduced solution quality and an inaccurate period are reflected in the median and percentile bands, which exhibit a noticeable deviation in amplitude and phase after three periods compared to the true trajectory.