Towards Integration of PeN-ODEs in a Modelica-based workflow

Andreas Hofmann[1] Lars Mikelsons[1]

[1]Chair of Mechatronics, University of Augsburg, Germany,
{andreas.hofmann,lars.mikelsons}@uni-augsburg.de

Abstract

Hybrid modeling – the combination of first-principle models and machine learning – offers the potential to increase model accuracy while reducing modeling effort. Although approaches for creating hybrid models from system simulation models exist, the unique characteristics of Modelica-based, object-oriented models – such as modularity and reusability – can, as of today, not be utilized. In this contribution, we explore approaches for bridging this gap to enable the use of hybrid models with Modelica. Key challenges of architecture definition, training environment and reintegration of the trained machine learning parts into a Modelica model are addressed. To illustrate our approach, we present a case study involving a SCARA robot. This example demonstrates a partially integrated workflow for hybrid modeling, intended to serve as a foundation and motivation for further research.

Keywords: hybrid modeling, PeN-ODE, SciML, NeuralFMU, Julia, machine learning

1 Introduction

1.1 Motivation

In recent years, data-driven models, i.e. machine learning (ML) models, and hybrid models, which combine first-principle modeling (FPM) and data-driven approaches, have gained wide attention in the field of system simulation. Data-driven models promise to eliminate efforts for traditional modeling tasks, while additionally increasing the fidelity of models based on real-world data. Alongside those benefits, however, data-driven models are limited by their high demand for data to approximate the underlying physics, their black-box nature, and limited extrapolation capabilities, thus restricting their application. While early architectures recurrently mapped states from a point in time to the next one, the more sophisticated approach of the NeuralODE paper (Chen et al. 2018), which gained a lot attention, confined data-driven models to the system dynamics and combined them with well-known numerical solvers.

Hybrid models, in contrast to their purely data-driven counterparts, try to incorporate ML-approaches into models that are built upon laws of physics and well-established empirical approaches. Since those models aim to only learn unmodelled physical effects, their data requirements are typically way more economic. Also, extrapolation capabilities and explainability of hybrid models are usually better compared to solely data-driven models due to the underlying FPM nature.

While the term NeuralODE is widely established in the field of data-driven approaches, various names and architectures for hybrid models abound, such as NeuralFMU (Thummerer, Kircher, and Mikelsons 2021), Universal Differential Equation (Rackauckas, Ma, et al. 2021), et cetera. Hence, since there is no common wording for hybrid models, their application, especially in industrial scope, is shortened. Therefore, a concise definition for hybrid models (for system simulation) has been developed in the the ITEA OpenSCALING project[1], based on earlier definitions by Thummerer and Mikelsons (2023), Thummerer, Kolesnikov, et al. (2023) and Kamp, Ultsch, and Brembeck (2023). The definition will be presented in the next section.

Modelica's object-oriented nature makes it especially well-suited for hybrid modeling, as it enables efficient reuse of models. On one hand, integrating machine learning models into component equations introduces a high degree of flexibility. This allows for modification of arbitrary mathematical relationships, like algebraic equations and state derivative expressions. Furthermore, a trained hybrid component can be reused across different modeling scenarios. For example, it is often advantageous to model an entire test rig, including both the device under test and the dynamic behavior of the testbed components. Using recorded data, a hybrid model of the device under test can be trained, subsequently extracted from the testbed model, and then embedded into a high-fidelity model of another system. This reusability helps justify the considerable effort involved in the training phase of hybrid modeling. On the other hand, object-oriented models retain information about algebraic relationships, which facilitates the direct integration of measurement data. As long as a relationship between the measured and augmented quantities exists, the model can be trained. This approach is particularly valuable when working with field data from systems in operation, where typically only a limited set of dedicated signals is available.

However, developing hybrid models from Modelica-based systems and reintegrating them into a Modelica-centered workflow presents several challenges. Key obstacles include the training process – typically performed

[1]project page: https://openscaling.org/

within machine learning frameworks developed in Python or Julia – and the integration of Neural Networks, which are among the most commonly used machine learning models, into the Modelica environment.

1.2 Outline

In the following section, a short overview of the PeN-ODE architecture and a further partitioning based on training strategies is provided. Following that, main challenges for building and training PeN-ODEs and utilizing those in a Modelica-based workflow are depicted. In the subsequent chapter, a workflow for PeN-ODEs is demonstrated, utilizing an object-oriented model of a Scara-Robot, however starting from the Julia programming language, avoiding the transfer of the model from a Modelica-based tool. Finally, the paper closes with an outlook on further developments and challenges.

2 Definition of PeN-ODE

As mentioned before, several methods for combining first principle models and machine learning have been available for many years. To make the concept of hybrid models accessible for wider industrial application, a common definition for hybrid models, or at least for the distinct class of system simulation, has been developed within the ITEA4 OpenSCALING project under the name PeN-ODE. Although this concise definition is limited to a model architecture, a further distinction, based on training strategies, can be made, which facilitates overall communication on required data, et cetera.

2.1 PeN-ODE Architecture

A PeN-ODE, short for Physics-enhanced Neural Ordinary Differential Equation, is an ODE[2] model that combines FPM and ML models, typically Neural Networks. As illustrated in Figure 1, no particular restrictions are imposed on the interaction between the FPM and ML subsystems, allowing maximum flexibility in tailoring the hybrid model to individual use cases.

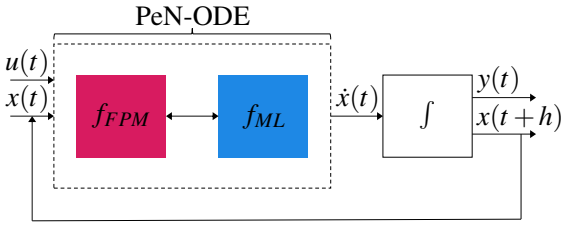

Figure 1. Architecture of a PeN-ODE with physical subsystem (f_{FPM}) and machine learning part (f_{ML}) within the solution process.
$x(t)$ - system states
$u(t)$ - inputs
$y(t)$ - outputs
h - solver step size

While this definition might seem arbitrary at first, it clearly differentiates the PeN-ODE from NeuralODEs and other data-driven approaches, where the complete dynamic of the system is represented using Neural Networks. Nonetheless, other common approaches for hybrid models are fully or partially incorporated: NeuralFMUs, the combination of Functional Mockup-Units and Neural Networks are a subset of the PeN-ODE, where the FPM is represented using the FMI standard. Universal Differential Equations (UDEs) describe a rich framework for ODEs, PDEs, SDEs, etc. and allow to build and train hybrid models but also models without any FPM (therefore similar to NeuralODEs). Also, training strategies such as Physics Informed Neural Networks are featured in the UDE framework. In this context, PeN-ODEs cover a dedicated subset of UDEs, focusing on the relevant parts for system simulation. There are far more approaches that fit into the PeN-ODE definition, though not necessarily from the field of technical system, such as Integrated Neural Network(Su et al. 1992) or methods without dedicated names (Quaghebeur, Nopens, and De Baets 2021), (Oliveira 2004). Sorourifar et al. (2023) also refer to their architecture as "Physics-Enhance Neural Ordinary Differential Equation", combining first-principle modelling and Neural Networks in chemical reaction systems.

Please note that in the PeN-ODE definition no training strategies, i.e. the training of the Neural Network, are explicitly defined. As a result, a pre-trained ML model augmenting an FPM is a valid PeN-ODE model.

2.2 Partitioning of PeN-ODE by training strategies

Although the PeN-ODE definition imposes no restrictions on the training process itself, different training strategies come with varying requirements for training data. Accordingly, alongside the definition of the PeN-ODE architecture, two distinct training strategies are delinated. It is important to note that, following successful training and integration, the specific training strategy used for the PeN-ODE becomes indistinguishable in the final model.

Closed-loop trained PeN-ODE

The first training strategy involves parameter optimization of the ML part by computing gradients through the solution of the ODE. Because this requires calculating gradients across the transient behavior of the system, aspects such as time and state events, as well as the numerical solution process itself, must be taken into account. For further details we refer to Thummerer, Olsson, et al. (2025), though focusing on NeuralFMUs.

In addition to the commonly used method of automatic differentiation, mathematical techniques, such as the Continuous Adjoint method or the Forward Sensitivity analysis for ODEs, have regained attention in recent years. The first approach performs the gradient calculation by tracing every operation on the discretized system, often referred to as "discretize-then-optimize". In contrast, the latter constructs an additional set of differential equations which are

[2]including DAEs and discretized PDEs

DOI 10.3384/ecp218435

solved alongside the original system. Since gradients are derived from the time-discrete solution values, it is often referred to as "optimize-then-discretize". A comprehensive review covering the details of gradient computation for ODEs can be found in (Sapienza et al. 2024).

The cost function of the optimization problem can aggregate any related quantity of the ODE system associated to the equations under consideration, i.e. alongside the states also algebraic quantities. Hence, considering the substantial effort for gradient computation, the constraints on available measurements from the (real-world) data are typically low. In Figure 2 the training strategy for closed-loop training is depicted.

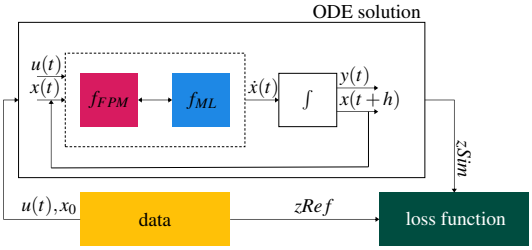

Figure 2. Closed-loop training procedure for loss calculation
$x(t)$ - system states
$u(t)$ - inputs
$y(t)$ - outputs
$zSim$ - relevant solution values from simulation
$zRef$ - reference solution values from data
x_0 - initial states
h - solver step size

Open-loop trained PeN-ODE

In contrast to the closed-loop training strategy, the open-loop procedure does not consider a transient cost function accumulating over the ODE solution. Instead, it uses state derivative quantities and algebraic variables to train the ML part based on the right-hand side (RHS) of the ODE, cf. Figure 3.

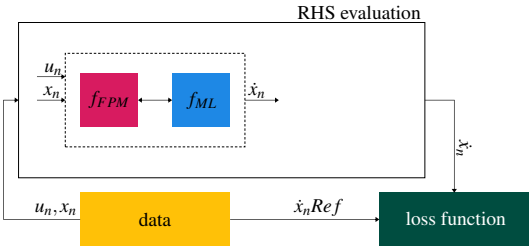

Figure 3. Open-loop training procedure for loss calculation, based on evaluation of the RHS of the ODE system
x_n - system states at time n
u_n - inputs at time n
$\dot{x}_n$ - state derivatives at time n
$\dot{x}_nRef$ - reference data at time n

As a result, the computational effort for gradient calculation is reduced and is typically handled via automatic differentiation. However, this approach places higher demands on the training data, as all relevant quantities must be explicitly available. Moreover, deviations during training on derivative level typically accumulate to increasing errors due to integration in the simulation of the trained PeN-ODE.

Although Figure 3 illustrates a joint evaluation, it is important to note that the machine learning component can also be trained independently. This corresponds to conventional data-driven approaches, as depicted in Figure 4.

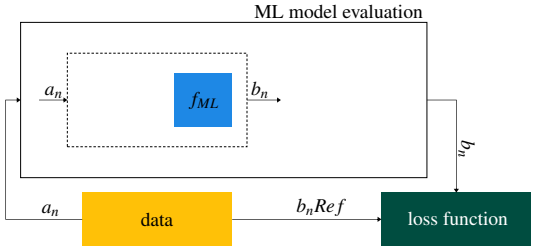

Figure 4. Open-loop training procedure for loss calculation, based on solely evaluating the ML model.
a_n - ML input at time n
b_n - ML output at time n
b_nRef - reference data at time n

3 Challenges for derivation and training of Modelica-based PeN-ODEs

As briefly mentioned in the introduction, utilizing PeN-ODEs in Modelica poses various challenges, which will be examined in more detail below. The primary challenges can be summarized as follows:

- **Training of the hybrid model:** Bridging the gap between Modelica-based modeling and common machine learning training frameworks, such as Julia
- **PeN-ODE architectures:** Limiting access to available signals and defining appropriate PeN-ODE architectures within the modular, object-oriented structure of Modelica.
- **Representation of hybrid architecture in Modelica:** Methods for incorporating trained ML components into Modelica models in a way that preserves their functionality and compatibility.

3.1 Training of the hybrid model

The first challenge in establishing an integrated workflow for PeN-ODEs in Modelica is bridging the gap to the ML training environment. Python and its extensive ecosystem of ML frameworks and libraries are indisputably the most common environment for machine learning tasks. Nevertheless, noting that many other suitable programming languages exist, we want to focus in the following passage on the Julia programming language. Julia offers a wide ecosystem for scientific machine learning (SciML) and does provide some initial approaches for bridging ML training environment and Modelica-based tools.

Tinnerholm et al. (2021) present a reimplementation of the OpenModelica compiler in Julia, called OM.jl, that can generate models for the Julia packages DifferentialEquations.jl (Rackauckas and Nie 2017), which provides methods for efficiently solving a wide range of differential equation problems, such as ODEs, DAEs, or SDEs. Additionally OM.jl can generate models for the Julia-specific environment for object-oriented modeling, ModelingToolkit.jl (Ma et al. 2021). In an additional step, similar to a Modelica compiler, ModelingToolkit symbolically processes the system to an executable set of differential-algebraic equations. Unfortunately, the corresponding Github-repositories (OM.jl, OMBackend.jl, ...) have not been maintained for years.

Similar to ModelingToolkit.jl, Modia (Elmqvist et al. 2021) offers capabilities of object-oriented modelling and simulation in Julia. Being strongly embedded into the Modelica community, the developers see Modia as an environment to test new algorithms and extensions towards further adaptions of Modelica. A translator has been developed by Otter, Elmqvist, et al. (2019), allowing to source-to-source translate a (subset) of the Modelica language to Modia. Unfortunately, the repository presented in their paper is no longer available. Bruder and Mikelsons (2021) utilized Modia to build an equation-based, hybrid non-linear single track model. However, like OM.jl, Modia is still based on an older version of Julia, released in Nov. 2021.

The Base Modelica initiative (Kurzbach et al. 2023) provides another approach for translating arbitrary, though only flattened, Modelica models to Julia. A dedicated package in the Julia SciML ecosystem exists, named BaseModelica.jl[3] , promising to parse and convert the Base Modelica model to the previously mentioned object-oriented modeling environment ModelingToolkit.jl. Please note that the initial Modelica change proposal for Base Modelica is currently in development. Once available, the pipeline could be an auspicious way of translating Modelica models to suitable simulation models for hybrid training in Julia. However, as of today, it remains unclear whether ModelingToolkit.jl is capable of supporting all relevant features of Modelica. Furthermore, in our experience, many Modelica models require specific numerical algorithms, which could cause further challenges.

Looking at the eFMI standard (Lenord et al. 2021), a representation reminiscent to the Algorithm Code, however without implemented solver arithmetic, could pose as a valid mutual baseline, for code generation to various languages and tools. From a Modelica compiler perspective, this would represent the hybrid[4] DAE system after the symbolic transformation. In contrast to Base Modelica, this representation would not require any additional transformation process. However, numerical challenges, as mentioned before, would still not be resolved by this intermediate representation.

Utilizing Julias' SciML ecosystem, while using the Functional Mockup Interface instead of Modelica directly, is possible through the packages FMIFlux.jl[5] and FMI.jl[6], which build the foundation of NeuralFMUs. FMI.jl allows to load and simulate FMUs in Julia, while FMIFlux.jl manages the gradient calculation through the FMU model. However, since all object-oriented information from a Modelica model is removed during FMU generation, it is not feasible to unroll the ML model of the trained NeuralFMU back to the object-oriented components.

Although we focused this section on model transfer from Modelica to Julia, the Enzyme project offers a solution to calculate the relevant gradients directly on LLVM level, see (Moses and Churavy 2020). LLVM supports a wide range of languages, however, changes to existing Modelica-based tools, that we cannot evaluate, might be needed. Furthermore, dedicated training frameworks, possibilities to define ML models, etc., within the Modelica tools would still be missing.

3.2 PeN-ODE architectures

Many hybrid model architectures, such as the NeuralFMU with its FMU foundation, grant access to the complete set of states, state derivatives, and algebraic quantities of the FPM. Consequently, hybrid models using arbitrary FPM signals in the ML model can be derived. Therefore, NeuralFMUs can be reused within other simulation models as a single component. However, separating the ML part from the physical model and re-introducing it in the original model and, therefore, also reusing partial models is not feasible. From an object-oriented modeling perspective, the use of arbitrary model quantities contradicts the paradigm of knowledge encapsulation, see (Cellier 1996). Consequently, the ML part of a hybrid component in an object-oriented model can only be supplied with internal quantities or existing interfaces of the model or additional signals must explicitly be exposed through model adaptions. Since encapsulation of knowledge enables the composition of hierarchical models, relying solely on available quantities renders reusing trained individual components possible.

As already stated in the PeN-ODE definition, arbitrary architectures for interaction between the FPM and ML part are applicable. Furthermore, ML models are adapted during their training phase and often optimized via hyperparameter optimization. Therefore, it is useful to only define the minimum necessary architectural parts, typically input and output features, within Modelica and shift all other architecture considerations to the ML side, where their optimization can take place. However, a Modelica compiler needs to know the relations of quantities to perform the symbolic transformation. The Modelica lan-

[3] https://github.com/SciML/BaseModelica.jl

[4] in this very case of this paper hybrid refers to continuous-discrete rather than to combination of FPM and ML

[5] https://github.com/ThummeTo/FMIFlux.jl

[6] https://github.com/ThummeTo/FMI.jl

DOI 10.3384/ecp218435

guage specification (Modelica Association 2023) provides the constructs of function inlining, which enables a placeholder functionality sufficient for the previously described considerations. Still, different types of augmenting the FPM equations exist.

The easiest way for setting up a hybrid architecture by place holder in a Modelica models is parallel execution on equation level, see Listing 1. Missing information in the equation is compensated by adding the function *fun_1*, which will later be replaced by some ML architecture. Only relevant inputs and outputs are provided in the Modelica model. Through the annotation "Inline=false" the function is not symbolically processed in the compiler and can be replaced during code generation.

Listing 1. parallel architecture on equation level

```
model SimpleParallelArchitecture
    function fun_1 "dummy function"
        input Real b, c, d;
        output Real dx;

    algorithm
        dx = b*c+d; // some operations that
        // will be replaced
    annotation(Inline=false);
    end fun_1;

    Real x "state variable";
    Real a,b,c,d;
    ...

equation
    ...
    // adding missing parts of the equation
    // by a Neural Network
    der(x) = a * x +  fun_1(b,c,d);
    ...
end SimpleParallelArchitecture;
```

The parallel architecture offers a coherent augmentation of a physical equation. However, to enable arbitrary architectures on ML side, further adaptions to existing model equations are required. This is done by splitting the relation of interest into two parts, as shown in Listing 2.

Listing 2. universal architecture on equation level

```
model UniversalArchitecture
    Real a "some algebraic qoi";
    Real a_fpm "the known relation of the
        qoi";
    Real b,c,d,x; // some additional
        variables
    ...
equation
    ...
    // a = b*x; // original equation
    a_fpm = b*x;
    a = fun_2(a_fpm, c,d);
    a = sin(time);
end UniversalArchitecture;
```

A detailed discussion on combination of FPM and ML, though in the context of NeuralFMUs, can be found in (Thummerer and Mikelsons 2025).

The relation between the algebraic variable a and the quantities b and x in Listing 2 shall be enhanced by some data-driven approach. Therefore, the first principle knowledge is rearranged into a new variable *a_fpm*. The original equation is replaced by the place holder function *fun_2*, that utilizes the existing knowledge of *a_fpm* and further employs the information from variables c and d. On ML side arbitrary combinations for interaction of the variables are possible, including the parallel architecture described before. Since the augmentation takes place on equation level rather than variable level, non-linear systems can arise from the function inline annotation during the symbolic transformation of the model. This is depicted in Listing 2 by explicitly providing a value to variable a, since time is the independent variable of a Modelica model. Solving the nonlinear system introduces additional evaluations during simulation. This is necessary because the placeholders used at this stage do not yet contain the actual relationships, which are only incorporated at a later point in the workflow. Obviously, on the machine learning side, the training framework must be capable of solving the nonlinear system and computing the corresponding gradients. To our current understanding, there are no further limitations imposed by Modelica itself, since the language specification has a rich extensiveness, like providing derivative annotations for the place holder function, et cetera.

If the architecture of the ML model is known before training, an implementation in Modelica could be possible. Some approaches will be discussed in the next section. However, since all equations from the ML model are processed by a Modelica compiler, additional effort is added to the symbolic transformation. Furthermore, the benefit of the symbolic representation of the ML model within Modelica remains unclear.

With experience, at least with ModelingToolkit.jl, we would recommend to only define the minimum required interfaces in Modelica and process all other parts in the ML environment. Having trainable parameters available as arrays, contrary to current Modelica compiler implementations, provides a convenient workflow in combination with hyperparameter optimization.

3.3 Representation of hybrid architecture in Modelica

As stated before, the final ML model is typically derived throughout the training in a hyperparameter optimization. Once the final structure has been established, the ML model and parameters must be reintegrated to the Modelica environment. Since Neural Network are today's dominating approach, we will limit approaches to this ML model type.

The most naive approach is implementing the corresponding tensor operations directly as a Modelica model, since the Modelica language provides features for multidimensional arrays. However, this is typically limited to simple architectures such as multilayer perceptions

(MLPs). A more sophisticated approach, while keeping Modelica language features, is provided in the NeuralNetwork Library[7] developed at Hochschule Bielefeld , that allows to automatically generate a model from Tensorflow. However, since array equations are, as of today, scalarized throughout the model transformation in a Modelica compiler, large Neural Networks might cause high compilation times or even overstrain available computer memory. Certainly, algorithms for keeping array structures throughout the symbolic pipeline of a Modelica compiler have been developed and will mitigate those issues, e.g. (Abdelhak, Casella, and Bachmann 2023), (Otter and Elmqvist 2017), (Fioravanti et al. 2023). Nonetheless, as stated before, the benefit of symbolic neural network representation in a Modelica model remains unclear.

Alternatively, standardized formats for exchanging deep learning models among tools, such as the Open Neural Network Exchange (ONNX) or the Neural Network Exchange Format (NNEF), could be utilized, for example the SmartInt Library[8] , which provides ONNX support using Modelica's "external C" feature. Therefore, Neural Networks with high number of parameters are be applicable, without any impact on a Modelica compiler.

3.4 Summary on Findings

There exist a wide range of different solution approaches which, in combination, help to prepare, create, and implement PeN-ODEs in Modelica. Modelica language features can be used to define interfaces in a convenient way and libraries and tools are available to embed ML models in Modelica. Nonetheless, the transfer of the (processed) Modelica model to a training environment such as Julia, poses an unsolved challenge. Once Base Modelica, is available, its application with ModelingToolkit.jl should be further investigated. Having a new implementation, positioned between Base Modelica (flattened) and the Algorithm Code from eFMI (including solver arithmetic) could pose a useful new representation, also for other applications. Further experiments with Modia or OM.jl might enable a first integrated workflow bridging the gap between Modelica and Julia, at the cost of limitations from the machine learning environment, due to their dependency on an old version of the Julia language.

We are confident, once PeN-ODEs have proven their usefulness, even against the current issues, a joint solution for bridging the gap can be achieved, suitable for users and tool vendors at the same time.

4 Scara-Robot Example

Based on the previously described solution approaches for integration PeN-ODEs in a Modelica workflow, we implemented an example application featuring a SCARA robot. However, since there exists no up-to-date solution for bridging the gap from Modelica to Julia while preserving the object-oriented structure of the model, we bypass this step in that we start from an object-oriented model within ModelingToolkit.jl. We hope that this example will encourage people to approach the findings from the previous chapter and to apply the idea of Modelica-based PeN-ODEs.

4.1 Overview on the SCARA Robot Example

The example is a modification of the NeuralFMU workshop (Thummerer and Mikelsons 2024) adapted for object-oriented application. A SCARA robot is used to write text on a sheet of paper by actuating its two revolute joints. A marker pen is mounted at the end effector, which can be pressed against the paper for writing or lifted for non-writing movements. The joints are actuated by DC motors and controlled using simple proportional controllers. An inverse kinematics model converts tabular end-effector position data – linearly interpolated between setpoints – into the corresponding joint angles.

The SCARA robot is modeled as 2D mechanical system and the writing process is modeled by applying a defined normal force, rather than implementing actual pen movement and collision considerations. Most importantly, no friction between the paper and the pen is model, but is supposed to be learned within the PeN-ODE. An equivalent Modelica model is illustrated in Figure 5. As reference data, simulation results from the robot containing Stribeck friction between end effector and paper, writing the words "train" and "validate" are available. To render a realistic scenario, we will only use motor currents as the available measurement outputs, since those signals are commonly available quantities and most importantly contain the friction force resp. acceleration information.

4.2 Model generation

As already mentioned at the beginning of this section, we employ ModelingToolkit.jl, to actually start from an object-oriented simulation model of the SCARA robot. Apart from own model implementations, models from the Modeling Toolkit Standard Library[9] were employed. Utilizing the Modelica PlanarMechanics library (Zimmer 2012) as a blueprint, necessary components for the mechanical system were implemented in ModelingToolkit.jl. All files reproducing the example can be found in the corresponding github repository[10]

As proposed previously, we insert a place holder function in the end effector model and consider only the known quantities within the component, i.e. end effector velocity in x- and y-direction and the normal force to the paper plane, to allow for the reuse of the trained subsystem, see Listing 3. Please note that "$\sim$" is used as the equality sign in ModelingToolkit.jl and the Modelica "der()"-operator corresponds to the "D()"-operator in the listing. NN_dummy refers to the place holder function that will be replaced after code generation. Its implementation is

[7] https://github.com/AMIT-HSBI/NeuralNetwork
[8] https://github.com/xrg-simulation/SMArtInt
[9] https://github.com/SciML/ModelingToolkit.jl
[10] https://github.com/AndreasHofmann217/ScaraRobotExample

DOI 10.3384/ecp218435

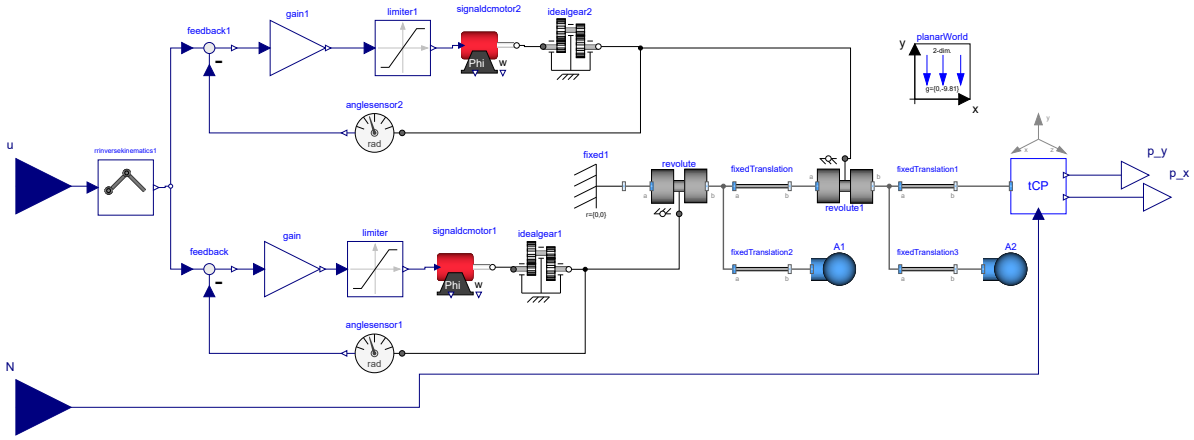

Figure 5. Illustration of the SCARA robot as Modelica model

similar to the previously mentioned workflow with annotation "Inline=false" from the Modelica specification.

ModelingToolkit.jl does generate an ODE system for models that can be represented by a structure incidence matrix with lower triangular form after transformation and a DAE system in mass matrix form otherwise. For the SCARA robot model a DAE system is generated, whose code is further adapted by us. The mass matrix, parameter information, and modified right-hand side functions are exported to a new Julia file, that will later be used within our training pipeline.

Listing 3. trainable end effector model

```
@mtkmodel TrainableTCP begin
    @variables begin
        x(t)
        y(t)
        vx(t)
        vy(t)
        f(t)[1:2]
    end
    @components begin
        frame_a = Frame()
        zForceIn = Blocks.RealInput()
        posXOut = Blocks.RealOutput()
        posYOut = Blocks.RealOutput()
    end
    @equations begin
        frame_a.tau ~ 0.0
        frame_a.fx ~ f[1]
        frame_a.fy ~ f[2]
        frame_a.x ~ posXOut.u
        frame_a.y ~ posYOut.u
        frame_a.x ~ x
        frame_a.y ~ y
        D(x) ~ vx
        D(y) ~ vy
        f ~ NN_dummy(1.0,vx,vy,zForceIn.u)
    end
end
```

4.3 ML model and PeN-ODE training

For learning the friction forces from the applied normal force and the velocities at the end effector, a Neural Network is employed as ML model. It is built using an MLP with three hidden layers, 32 neurons and tanh as the activation function. To scale input features (velocity in x- and y-direction, normal force onto the paper) as well as the output features (friction forces in x- and y-direction), a layer adding z-score standardization resp. another layer providing inverse transformation for the output features are added. However, those layers do not contain fixed values for means and standard deviations but are rather trainable parameters as well. As a last measure, a trainable gate, initialized with start value of zero is added. This guarantees that the randomly initialized network does provide stable output during the first training steps. The complete architecture is depicted in Figure 6.

The model is trained using a single trajectory, with the word "train" written by the SCARA robot, cf. Figure 7

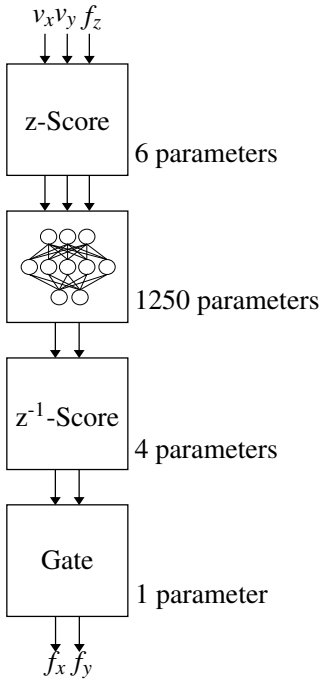

Figure 6. Employed ML model for the SCARA robot and the individual number of trainable parameters for the individual parts

Currently, only single-step solvers can be employed for the hybrid DAE of the SCARA during gradient calculation[11]. Therefore, we use Rodas5P, a fifth order Rosenbrock method from within DifferentialEquations.jl, as numerical method.

The trajectory is split into samples of 50 ms length and gradients are calculated over the solution of randomly selected single samples. The sampling is done for various reasons. On the one side, it helps to increase training speed, since gradients over a short elapsed time can be calculated faster and have lower requirements for computer hardware. Additionally, using stochastic examples helps to better progress in the beginning of the optimization, while calculating only gradients for a few examples, cf. (Goodfellow, Bengio, and Courville 2016). On the other side, since the calculated gradients combine all deviations across the simulated period, shorter time spans support capturing highly dynamic details. The Adam optimizer (Kingma and Ba 2015) with default settings was used for updating parameters. For loss accumulating Mean Absolute Error (MAE) aggregation was chosen, considering motor currents as quantities as available signals, as stated above. As algorithm for deriving the gradients of the DAE ReverseDiffAdjoint from the package SciMLSensitivity.jl (Rackauckas, Ma, et al. 2021) was selected.

The system was trained for 5000 steps on a laptop with i7-11850H CPU, 32 GB RAM and Win11 as OS, which took about two hours. Details on the used Julia packages can be found in the corresponding GitHub repository.

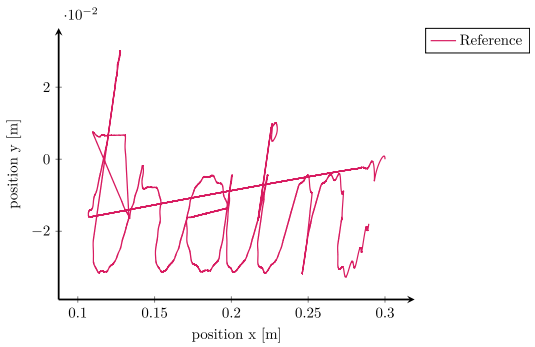

Figure 7. Training trajectory (end effector) for the SCARA robot (without distinguishing actually applied forces for writing)

4.4 Training results

Please note that no extensive effort, e.g. through hyperparameter analysis, was spent on the model. The results are purely derived to motivate and validate a PeN-ODE workflow, rather than to provide the best possible model outcome. Nonetheless, to anticipate some results of the trained model, the MAE for the combined model states is reduced by a factor greater than 4, while the Mean Squared Error (MSE) is reduced by a factor larger than 11, each for the chosen test trajectory.

As test data for the SCARA robot application, the word "validate" has been selected. In Figure 8, the model with and without training are compared to the reference data for the end effector position. The further the SCARA arms is extended, the more impact from friction is visible. The common error metrics MAE and MSE show a significant reduction in deviations. An overview for the aggregated model and individual states is collected in Table 1.

Table 1. Error metrics of the trained PeN-ODE compared to the original model (in brackets) for the test data trajectory.

state	MAE	MSE
complete model	7.76e-03	4.47e-04
$[-]$	(3.61e-02)	(5.19e-03)
$motor1.\varphi$	1.69e-04	1.17e-07
$[rad]$	(1.26e-03)	(2.87e-06)
$motor1.i$	1.40e-02	1.04e-03
$[A]$	(9.39e-02)	(1.61e-02)
$motor1.\omega$	5.59e-03	2.34e-04
$[rad/s]$	(1.62e-02)	(1.27e-03)
$motor2.\varphi$	3.40e-04	3.92e-07
$[rad]$	(1.67e-03)	(5.31e-06)
$motor2.i$	1.63e-02	9.58e-04
$[A]$	(7.60e-02)	(1.11e-02)
$motor2.\omega$	1.01e-02	4.55e-04
$[rad/s]$	(2.76e-02)	(2.70e-03)

A more detailed insight on the expressiveness of the PeN-ODE can be gained from the plots of the motor currents, see Figure 9 and Figure 10, as well as from the applied friction forces in the model, cf. Figure 11 and Figure 12. The model still holds some limitation w.r.t. force in x-direction, probably limited by the scalar gate. Furthermore, the friction forces during the start of the simulation, although no normal force is present, would require some further investigation on the training. Also, no further study on the impact of partly implementing physical effects, such as explicitly providing Coloumb friction, was conducted. This remains to be investigated in detail in the future. Yet, a first study on partially modelling friction, based on NeuralFMUs, was executed by Thummerer, Kolesnikov, et al. (2023).

From an object-oriented modeling perspective, the machine learning component can be seamlessly reintegrated into the end effector by replacing the placeholder function with the actual implementation containing the trained parameters. As previously noted, keeping these parameters independent of the compiler contributes to maintaining fast compilation times. Since only internal variables and inputs of the end effector component were utilized within the hybrid architecture, the component remains fully reusable in other applications – such as robots with different mechanical structures – without requiring further modifications.

[11] this is due to the discontinuous setpoint data and a bug during initialization of the DAE

DOI
10.3384/ecp218435

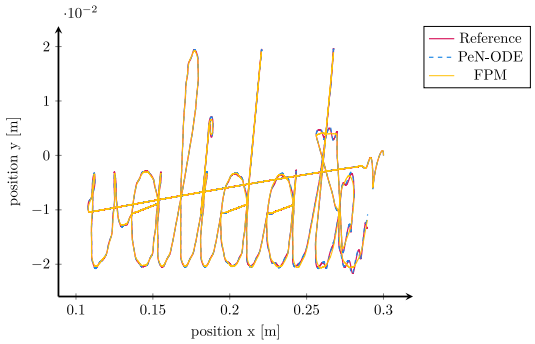

Figure 8. Comparison of the trained and untrained model writing the word validate (without distinguishing actually applied force)

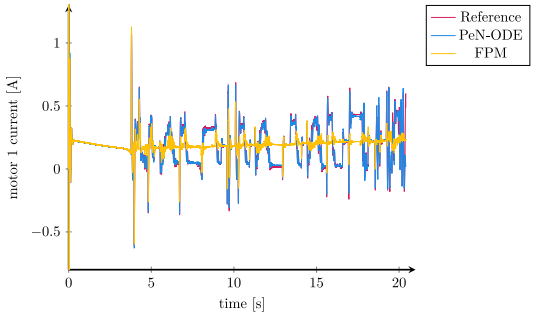

Figure 9. Comparison of Motor currents at Motor 1

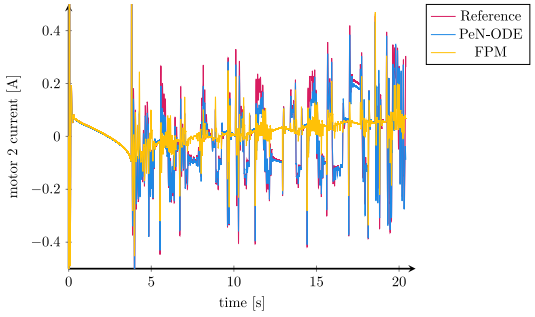

Figure 10. Comparison of Motor currents at Motor 2

5 Summary and Outlook

Although Modelica and the corresponding tools are natively incompatible with ML training environments such as the Julia programming language, many solution approaches for bridging the gap towards creating, training and applying of PeN-ODEs exist. Once a model is made available in Julia, all necessary steps can be applied, which was shown for the SCARA robot.

From the interfacing point of view, having an intermediate representation of the derived DAE system from a Modelica compiler, reminiscent of Base Modelica, would allow a universal starting point for arbitrary ML environments in Julia, Python or other languages but also for other applications. Furthermore, Base Modelica, once available, could offer a convenient way from Modelica to Julia, though it needs to be investigated if ModelingToolkit.jl is capable of supporting all relevant Modelica language features. Nonetheless, also tool vendor-specific implementations of Modelica compilers and simulation capabilities and particularities of their individual simulation runtimes might still pose challenges.

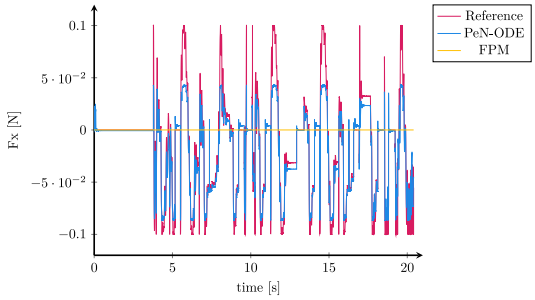

Figure 11. Comparison of friction force in x-direction

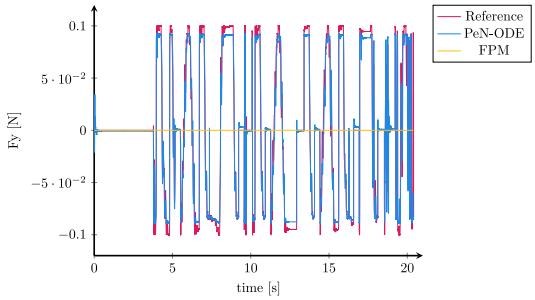

Figure 12. Comparison of friction force in y-direction

Apart from interfacing, new challenges, such as appropriate training strategies and best suitable ML models arise. While the example model was trained on a random selection of data samples, more sophisticated approaches such as dedicated scheduling algorithms for picking worst elements from the training set or different strategies, such as multiple shooting, growing horizon, et cetera, might accelerate and improve training.

In particular, the aspects of reusability of PeN-ODE components can facilitate the object-oriented approach from methods such as NeuralFMU, which in contrast does benefit from tool independence and bypasses the interfacing issues. A Modelica-based approach would allow to easily compose hierarchical systems of high fidelity PeN-ODEs.

Furthermore, the appropriate ratio of embedded physically-based white box models, such as known parts of friction, is an open research question for all hybrid models. Further studies are required to provide some indication, especially based on real-world problems.

Acknowledgements

This work has been partially funded through the ITEA4-project OpenSCALING (`www.openscaling.org`) by the German Federal Ministry of Education and Research (BMBF) under the grant number FKZ 01IS23062H.

References

Abdelhak, Karim, Francesco Casella, and Bernhard Bachmann (2023). "Pseudo Array Causalization". In: *Proceedings of the 15th International Modelica Conference 2023*. DOI: 10.3384/ecp204177.

Bruder, Frederic and Lars Mikelsons (2021). "Modia and Julia for Grey Box Modeling". In: *Proceedings of the 14th International Modelica Conference 2021*. DOI: 10.3384/ecp2118187.

Cellier, François (1996). "Object-Oriented Modeling: Means For Dealing With System Complexity". In: *Proceedings of the 15th Benelux Meeting on Systems and Control*.

Chen, Ricky T. Q. et al. (2018). "Neural Ordinary Differential Equations". In: *Proceedings in Advances in Neural Information Processing Systems 31 (NeurIPS 2018)*. DOI: 10.48550/arXiv.1806.07366.

Elmqvist, Hilding et al. (2021). "Modia – Equation Based Modeling and Domain Specific Algorithms". In: *Proceedings of the 14th International Modelica Conference 2021*. DOI: 10.3384/ecp2118173.

Fioravanti, Massimo et al. (2023-07). "Array-Aware Matching: Taming the Complexity of Large-Scale Simulation Models". In: *ACM Trans. Math. Softw.* DOI: 10.1145/3611661.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

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*. DOI: 10.5220/0012160100003543.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization." In: *ICLR (Poster)*. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2015.html#KingmaB14.

Kurzbach, Gerd et al. (2023). "Design proposal of a standardized Base Modelica language". In: *Proceedings of the 15th International Modelica Conference 2023*. DOI: 10.3384/ecp204469.

Lenord, Oliver et al. (2021). "eFMI: An open standard for physical models in embedded software". In: *Proceedings of 14th Modelica Conference 2021*. DOI: doi.org/10.3384/ecp2118157.

Ma, Yingbo et al. (2021). *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. arXiv: 2103.05244 [cs.MS].

Modelica Association (2023). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.6*. Tech. rep. Linköping: Modelica Association.

Moses, William and Valentin Churavy (2020). "Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients". In: *Advances in Neural Information Processing Systems*. Vol. 33. DOI: 10.48550/arXiv.2010.01709.

Oliveira, R. (2004). "Combining first principles modelling and artificial neural networks: a general framework". In: *Computers Chemical Engineering* 28.5. DOI: 10.1016/j.compchemeng.2004.02.014.

Otter, Martin and Hilding Elmqvist (2017). "Transformation of Differential Algebraic Array Equations to Index One Form". In: *Proceedings of the 12th International Modelica Conference 2017*. DOI: 10.3384/ecp17132565.

Otter, Martin, Hilding Elmqvist, et al. (2019). "Thermodynamic Property and Fluid Modeling with Modern Programming Language Constructs". In: *Proceedings of the 13th International Modelica Conference 2021*. DOI: 10.3384/ecp19157589.

Quaghebeur, Ward, Ingmar Nopens, and Bernard De Baets (2021). "Incorporating Unmodeled Dynamics Into First-Principles Models Through Machine Learning". In: *IEEE Access* 9. DOI: 10.1109/ACCESS.2021.3055353.

Rackauckas, Christopher, Yingbo Ma, et al. (2021). *Universal Differential Equations for Scientific Machine Learning*. arXiv: 2001.04385 [cs.LG]. URL: https://arxiv.org/abs/2001.04385.

Rackauckas, Christopher and Qing Nie (2017). "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia". In: *Journal of Open Research Software* 5.1.

Sapienza, Facundo et al. (2024). *Differentiable Programming for Differential Equations: A Review*. arXiv: 2406.09699 [math.NA]. URL: https://arxiv.org/abs/2406.09699.

Sorourifar, Farshud et al. (2023). "Physics-Enhanced Neural Ordinary Differential Equations: Application to Industrial Chemical Reaction Systems". In: *Industrial & Engineering Chemistry Research* 62.38. DOI: 10.1021/acs.iecr.3c01471.

Su, Hong-Te et al. (1992). "Integrating Neural Networks with First Principles Models for Dynamic Modeling". In: *IFAC Proceedings Volumes* 25.5. 3rd IFAC Symposium on Dynamics and Control of Chemical Reactors, Distillation Columns and Batch Processes (DYCORD+ '92), Maryland, USA, 26-29 April. DOI: 10.1016/S1474-6670(17)51013-7.

Thummerer, Tobias, Josef Kircher, and Lars Mikelsons (2021). "NeuralFMU: Towards Structural Integration of FMUs into Neural Networks". In: *Proceedings of 14th Modelica Conference 2021*. DOI: 10.3384/ecp21181297.

Thummerer, Tobias, Artem Kolesnikov, et al. (2023). "Paving the way for Hybrid Twins using Neural Functional Mock-Up Units". In: *Proceedings of the 15th International Modelica Conference 2023*. DOI: 10.3384/ecp204141.

Thummerer, Tobias and Lars Mikelsons (2023). *Using NeuralODEs in real life applications*. https://pretalx.com/juliacon2023/talk/EWL3LC/. Accessed: 2025-04-08.

Thummerer, Tobias and Lars Mikelsons (2024-08). *Scientific Machine Learning using Functional Mock-Up Units | Thummerer, Mikelsons | JuliaCon 2024*. https://www.youtube.com/watch?v=sQ2MXSswrSo.

Thummerer, Tobias and Lars Mikelsons (2025). *Learnable Interpretable Model Combination in Dynamical Systems Modeling*. URL: https://arxiv.org/abs/2406.08093.

Thummerer, Tobias, Hans Olsson, et al. (2025). "LS-SA: Developing an FMI layered standard for holistic efficient sensitivity analysis of FMUs". In: *Proceedings of the 16th International Modelica Conference 2025*.

Tinnerholm, John et al. (2021). "OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingTookit.jl". In: *Proceedings of 14th Modelica Conference 2021*. DOI: 10.3384/ecp21181109.

Zimmer, Dirk (2012). "A Planar Mechanical Library for Teaching Modelica". In: *Proceedings of the 9th International Modelica Conference*. DOI: 10.3384/ecp12076681.