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

Tobias Kamp[1] Christoff Bürger[2] Johannes Rein[1] Jonathan Brembeck[1]

[1]Institute of Vehicle Concepts, German Aerospace Center, {tobias.kamp, johannes.rein, jonathan.brembeck}@dlr.de
[2]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 accuracy. 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.

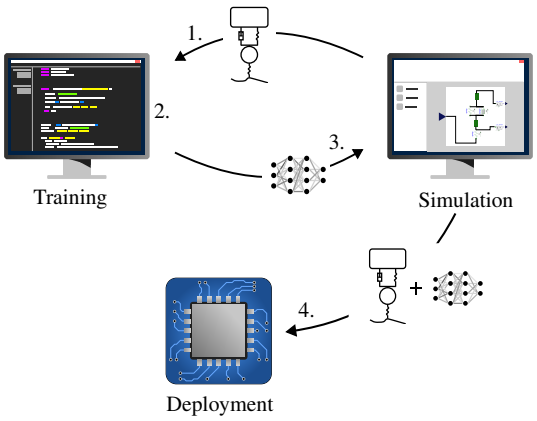


**Figure 1.** Development workflow of hybrid simulation models, including final deployment on the embedded target of a cyber-physical system.

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[1] model in Dymola[2], 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[3] in Python[4] or Julia[5]).

**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[6] (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[7] (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-

[1] https://modelica.org/language
[2] https://www.dymola.com
[3] https://pytorch.org
[4] https://www.python.org
[5] https://julialang.org
[6] https://fmi-standard.org
[7] https://onnx.ai


DOI 10.3384/ecp218555

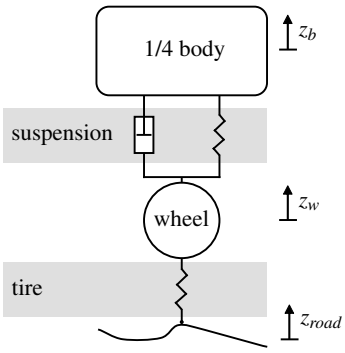


**Figure 2.** Scheme of the simple quarter vehicle model (QVM) comprising the body and wheel masses, a spring to represent the tire and a spring-damper pair for the suspension. The model input is the road height $z_{road}$; the vertical wheel and body positions $z_w$ and $z_b$ are the model outputs.

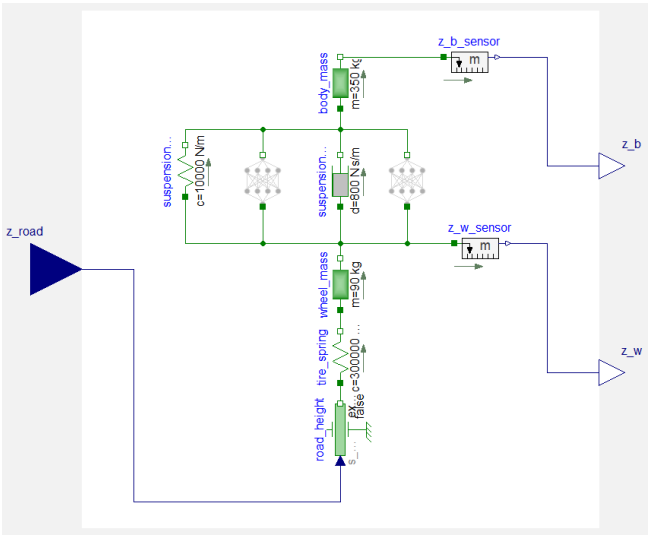


**Figure 3.** Modelica model of the neural QVM, i.e., the PeN-ODE. The two trained NNs capture non-linear effects of the suspension's spring and damper and are connected in parallel to the linear physics components. The NNs can be imported using generated C code from an ONNX representation (cf. Section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

bedded systems[8] (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[9] 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[10] 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[11] 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 (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

[8] https://www.efmi-standard.org

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

[10] https://github.com/AMIT-HSBI/NeuralNetwork

[11] https://github.com/kraiskil/onnx2c

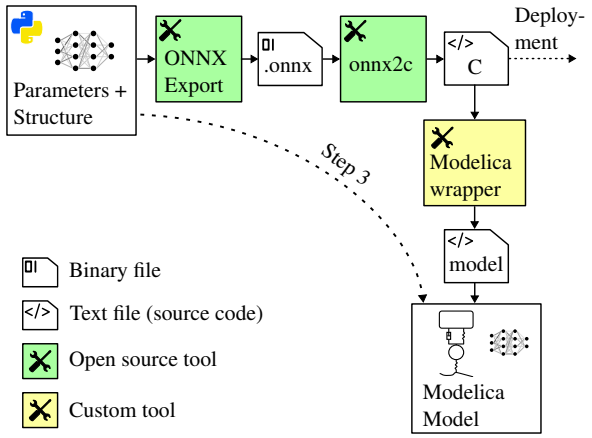


**Figure 4.** Scheme of the export toolchain using onnx2c to generate C code for ONNX models. Since many ML-frameworks already support ONNX export, we mark the ONNX export as openly available. Although the code onnx2c generates is suited for embedded application, there is no "ready-to-use" solution for 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 ..."[7]. 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[12] and PyTorch frameworks, MATLAB®[13] 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[11] compiler offers a remedy for users that can make use of plain C code. It is optimized for TinyML[14], 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>>",
```

[12] https://keras.io
[13] https://www.mathworks.com/products/matlab.html
[14] https://github.com/mit-han-lab/tinyml


DOI
10.3384/ecp218555

```
    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 microcontrollers, 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-ODE models as binary code FMUs without further dependencies.

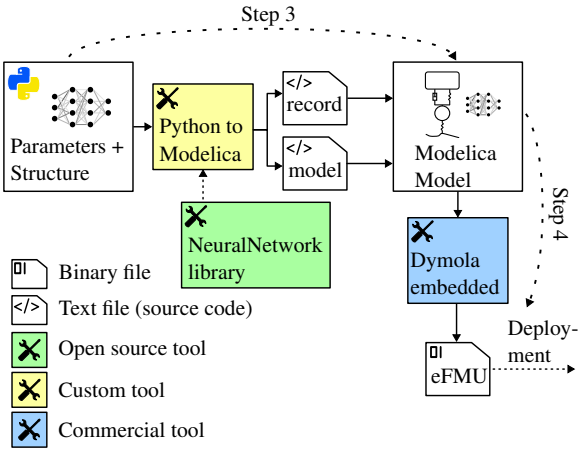


**Figure 5.** Scheme of the export toolchain employing the NeuralNetwork library. The Python to Modelica generator is a custom tool that allows direct Modelica code generation from Python. This approach leverages Dymola's existing symbolic facilities to find algorithmic solutions suited for embedded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

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 NeuralNetwork 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

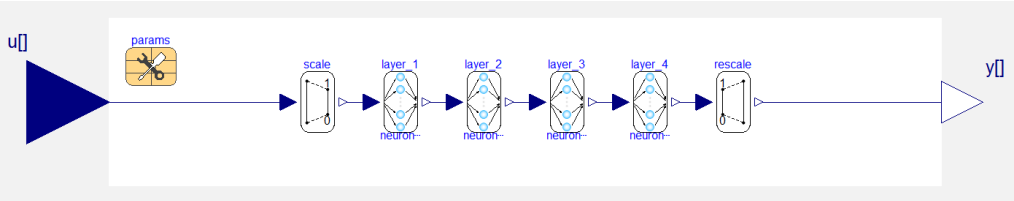


**Figure 6.** Generated, equation-based Modelica model of one of the trained NNs of the nQVM, showing its diagrammatic tensorflow using NeuralNetwork library components. The NN parameters (weights and biases) for each of the four dense layers ($16\times1$ – $16\times16$ – $16\times16$ – $1\times16$) 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[10]. 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"[8].

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, interfaces 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[15] to extend Production Code containers to become AUTOSAR[16] components.

It is noteworthy, that Software Production Engineering is also used as embedded code generation backend for No Magic Cameo Systems Modeler[17], a development environment for model-driven engineering, system architecture, system design, and software engineering based on SysML[18]. 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[19], import production code in MATLAB®/Simulink®[20] 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).

[15] https://www.3ds.com/products/catia/autosar-builder
[16] https://www.autosar.org
[17] https://www.3ds.com/products/catia/no-magic
[18] https://www.omgsysml.org
[19] https://www.cppcheck.com
[20] https://www.mathworks.com/products/simulink.html

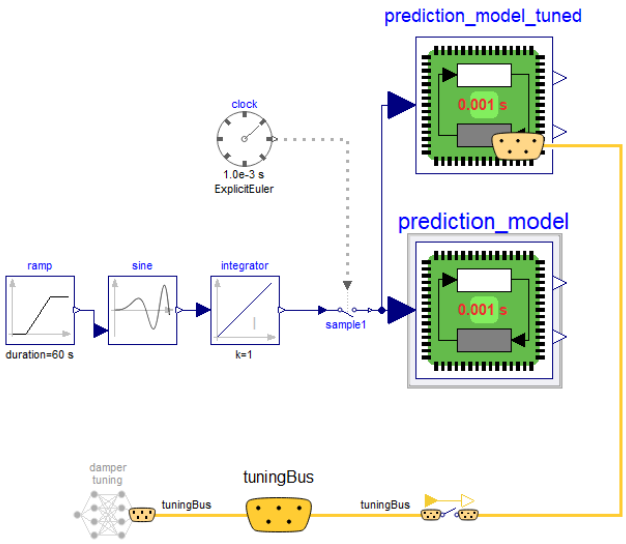


**Figure 7.** Recalibration SiL test in Dymola: The nQVM PeN-ODE is instantiated twice – once untuned and once tuned – via a generated Modelica wrapper backed by the eFMUs production code. The damper tuning component is manually implemented. It triggers recalibration at 2 s and applies updated NN parameters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

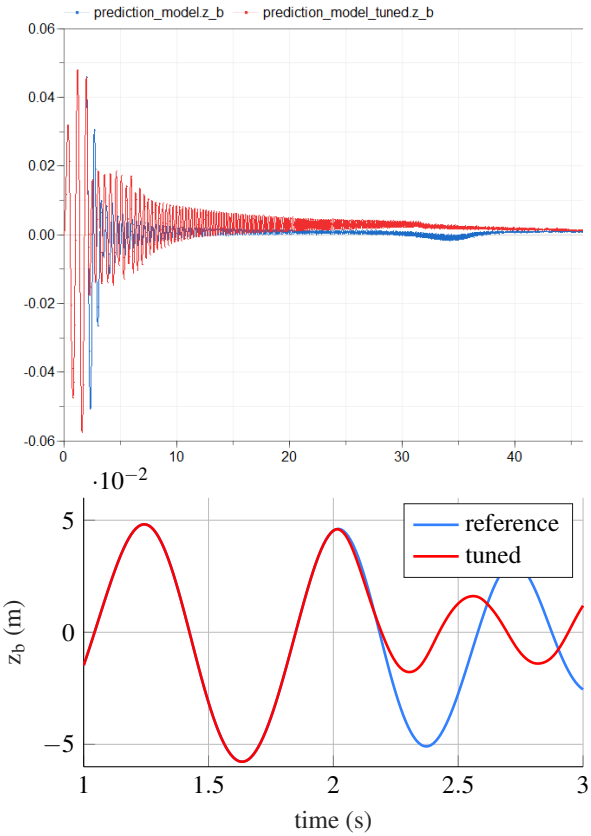


**Figure 8.** SiL simulation results of the body height $z_b$ for the untuned (blue) and tuned (red) nQVM. At 2 s simulation time, the recalibration is triggered and leads to a new system behavior.

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.

Figures 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 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 recalibraion 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[21] (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-

[21] https://itea4.org/project/openscaling.html


DOI 10.3384/ecp218555

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[22] (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 (16×1, ReLu), two hidden layers (16×16, ReLu), and the output layer (1×16, 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).

[22] https://ssp-standard.org

### 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[23] 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

[23] https://modelica.org/association

FMI in the simulation domain, and increasing support also in training environments (e.g., FMPy[24] for Python or FMI.jl[25] 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[26] offers a way to use ONNX models and TensorFlow[27] models exported as Lite Runtime[28] (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 perturbates 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[23], 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 nonlinear 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[29], grant number 2023-

[24] https://github.com/CATIA-Systems/FMPy
[25] https://github.com/ThummeTo/FMI.jl
[26] https://github.com/xrg-simulation/SMArtIInt
[27] https://www.tensorflow.org
[28] Formerly known as TensorFlow Lite (TFLite).
[29] https://www.vinnova.se

00969) and the German Federal Ministry of Education and Research (BMBF[30], grant number FKZ 01IS23062A) within the ITEA 4 Project Open standards for SCALable virtual engineerING and operation (OpenSCALING[21], 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.

[30] https://www.bmbf.de