Paving the way for Hybrid Twins using Neural Functional Mock-Up Units

Tobias Thummerer[1] Artem Kolesnikov[2] Julia Gundermann[2] Denis Ritz[3] Lars Mikelsons[1]

[1]University of Augsburg, Germany, {tobias.thummerer, lars.mikelsons}@uni-a.de
[2]ESI Germany GmbH, Dresden, Germany, {artem.kolesnikov, julia.gundermann}@esi-group.com
[3]Technische Universität Dresden, Germany, denis.ritz@tu-dresden.de

Abstract

Neural Ordinary Differential Equations (NeuralODEs) open up the possibility to enhance the modeling of dynamical systems, in terms of prediction quality and computation time, as well as shortened development time. Porting NeuralODEs, the combination of an artificial neural network and an ODE solver, to real engineering applications is still a challenging venture. However, we will show that *Neural Functional Mock-up Units* (NeuralFMUs), an evolved subgroup of NeuralODEs that contain *Functional Mock-up Units* (FMUs), are able to cope with these challenges. This paper briefly introduces to the topics NeuralODE and NeuralFMU and describes the procedure and considerations to apply this technique to a real engineering use case. Further, different workflows to apply NeuralFMUs dependent on tool capabilities and use case requirements are discussed. The presented method is illustrated with the creation of a *Hybrid Twin* of an hydraulic excavator arm, which features various challenges such as discontinuity, nonlinearity, oscillations and characteristic maps. Finally, we will show that the *Hybrid Twin* created on basis of measurement data from a real system gives more accurate results compared to a conventional simulation model based on physical equations (*first principle model*).

Keywords: NeuralFMU, NeuralODE, PeNODE, FMI, PhysicsAI, Hybrid Twin, Scientific Machine Learning

1 Introduction

In the following sections, short introductions to the used techniques are given.

1.1 NeuralODE

Since their introduction in 2018, *Neural Ordinary Differential Equations* (NeuralODEs) (Chen et al. 2018) are one of the key techniques for data driven modeling of physical systems. NeuralODEs consist of an *Artificial Neural Network* (ANN) that functions as the right-hand side of an *Ordinary Differential Equation* (ODE), together with an ODE solver to obtain the solution of the ODE, see Figure 1. Training such models on the ODE solution $\boldsymbol{x}$ requires (efficient) differentiation through the ODE solver by *Automatic Differentiation* (AD) or estimating sensitivities with *adjoint sensitivity analysis* (Bittner 1963). As almost any other machine learning model for learning dynamic systems like *recurrent neural networks* or *long short-term memory* networks in (Champaney et al. 2022), plain NeuralODEs need a significant amount of data and are only able to learn physical effects that are represented as part of this data. In real world engineering, there is often far more system knowledge available, which is only partially included or not included at all in the NeuralODE training data set. This knowledge can be used to drastically improve training, by incorporating it into the NeuralODE model itself, for example in the form of differential equations. The resulting structure, consisting of ANNs, differential equations and a numerical ODE solver, is further referred to as *Physics-enhanced Neural Ordinary Differential Equation* (PeNODE) or synonymously hybrid NeuralODE (see Figure 2). In this case, the ANN needs only to learn the missing physics that is not part of the system of differential equations. Compared to plain NeuralODEs, this allows for the use of smaller ANN

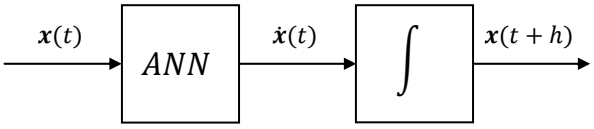

Figure 1. The topology of a NeuralODE. On basis of the system state $\boldsymbol{x}(t)$ the ANN computes the system state derivative $\dot{\boldsymbol{x}}(t)$, which is numerically integrated into the next system state $\boldsymbol{x}(t+h)$ by the ODE solver with step size h.

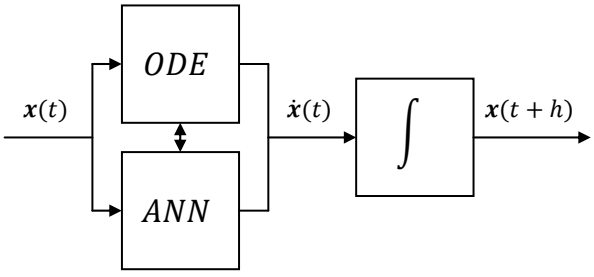

Figure 2. An example topology for a PeNODE. The system dynamics are determined by an ODE and an ANN, for which a variety of different interconnection topologies are possible.

topologies with less trainable parameters, which result in faster training times and often much better convergence. Further, introducing physical equations opens up to much better explainability regarding the learned process by the ANN, because a physical interpretation can be given for signals between ANN and physical equations. In practice, this *incorporated knowledge* might often be the part of the system that is well understood by the engineer and can therefore be modeled easily, for example the kinematics of an industrial robot, whereas the ANN has to learn the remaining physical effects that are challenging to model, like e.g. the friction behavior of the robot joints.

1.2 NeuralFMU

If the concept of a PeNODE shall be applied to real world engineering problems, another issue has to be faced: Physical models, designed in dedicated modeling tools, are not available in a symbolic representation of the equation system that can easily be used as parts of a PeNODE. Even if the symbolic ODE is accessible, large and complex systems often count thousands of equations, which is cumbersome to handle. Therefore, modeling PeNODEs on basis of large systems of equations, which are common in industrial applications, is not practicable out of the box. Fortunately, this can be solved by deploying the model foundation not by a system of equations, but a handy container for such: The *Functional Mock-up Unit* (FMU) (Blochwitz et al. 2011). The FMU type *Model-Exchange* (ME) provides an interface in analogy to an ODE system: On basis of the current time t, the system state $\boldsymbol{x}(t)$ and optional inputs $\boldsymbol{u}(t)$, the system dynamics $\dot{\boldsymbol{x}}(t)$ are calculated. Comparing to the PeNODE model, instead of combining an ODE with an ODE solver and ANN, a ME-FMU is used. This topology (see Figure 3) is referred to as *Neural Functional Mock-up Unit* (NeuralFMU) and was introduced in (Thummerer, Kircher, and Mikelsons 2021).

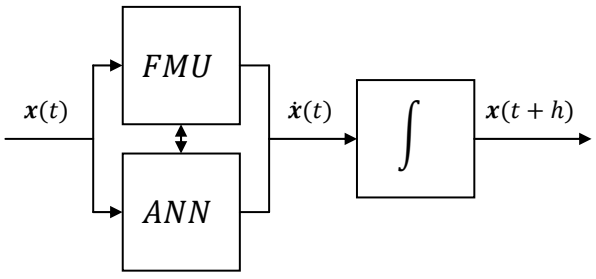

Figure 3. An example topology for a ME NeuralFMU. In this specific case, the system dynamics are determined by an ME-FMU and an ANN in parallel. In general, a wide variety of different interconnection topologies is possible.

Because determination of the loss function gradient for NeuralFMUs (and NeuralODEs in general) is computationally expensive compared to gradient determination in plain ANNs without ODE solvers, efficient differentiation is a necessity to achieve an economic training (and therefore development) time. The *Julia* programming language (Bezanson et al. 2014) offers some of the most powerful frameworks for AD that implement a variety of different approaches, therefore the first library for building and training NeuralFMUs called *FMIFlux.jl*[1] was implemented in this language. As a proof of concept for the applicability of NeuralFMUs in industrial applications, a vehicle longitudinal dynamics model for the prediction of an electric vehicle's energy consumption was enhanced in terms of result accuracy (Thummerer, Stoljar, and Mikelsons 2022). Further, the applicability to medical use cases was shown at the example of a hybrid simulation model of the human cardiovascular system (Thummerer, Tintenherr, and Mikelsons 2021).

2 Method

Deploying a NeuralFMU for a custom use case can be subdivided into three main tasks. First, the transfer of the *First Principle Model* (FPM) from modeling environment to the machine learning environment *Julia* (discussed in subsection 2.1). Second, the actual topology design and training of the NeuralFMU in *Julia* (subsection 2.2) and third, the reimport of the trained *Hybrid Model* (HM) from *Julia* into the original (or another) modeling or simulation environment (subsection 2.3).

2.1 From modeling environment to Julia

Because high performance differentiation is not available in most modeling tools, the FPM needs to be transferred from the original modeling tool into *Julia* (Thummerer, Stoljar, and Mikelsons 2022). This is achieved by using the FMU export functionality of the modeling tool to export an FMU and the *Julia* library *FMI.jl*[2] to import the FMU into *Julia*, see Figure 6 (step 1). After the import of the FPM FMU into *Julia*, the HM can be designed and trained.

2.2 Designing the topology

Basically, a wide range of topologies for NeuralFMUs are thinkable and designing a suitable one might not be intuitive. In the following, different aspects are highlighted and suggestions for decision-making based on requirements are given.

2.2.1 Sequential/Parallel

While the position of the numerical integrator in a NeuralODE is fixed, the positions of the FMU(s) and ANN(s) are not. Here, two main topologies can be distinguished. First, the elements can be connected sequentially, so one element is computing results on basis of intermediate results from another element, see Figure 4. A common use-case is an ANN, that corrects the system dynamics retrieved from an insufficient FMU model. Second, elements can be connected in parallel, so multiple elements

[1] `https://github.com/ThummeTo/FMIFlux.jl` (accessed on May 24, 2023)

[2] `https://github.com/ThummeTo/FMI.jl` (accessed on May 24, 2023)

DOI 10.3384/ecp204141

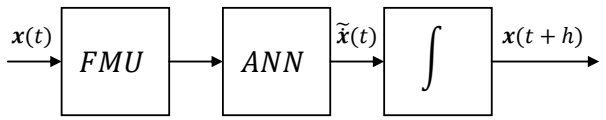

Figure 4. An example for a sequential NeuralFMU topology.

are computing results on basis of the same or different inputs, whereas the results need to be merged, as shown in the introduction in Figure 3. This approach is useful for example if the ANN needs to learn an effect that can be added to the existing dynamics (like many friction effects) or only a subpart of the state derivative vector shall be computed by the ANN, whereas the remaining part is determined by the ODE.

If no decision can be made on how the final effect will influence the system, e.g. because of lack of information regarding the unmodeled system part, both topologies can be used at once with minimal overhead: A topology using gates, as introduced in (Thummerer, Stoljar, and Mikelsons 2022), allows for continuous fading between a sequential and parallel interconnection of the ANN, see Figure 5. Note, that the gate parameters do not need to sum

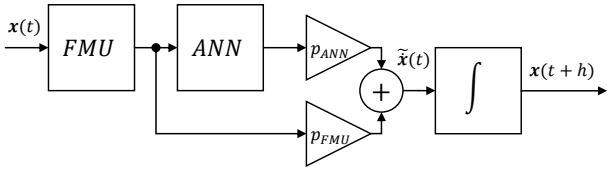

Figure 5. A NeuralFMU using gates. The parameters p_{ANN} and p_{FMU} control how much the ANN dynamics and FMU dynamics contribute to the final system dynamics $\tilde{\boldsymbol{x}}$. For vectors of signals, gate parameters can be used for all signals at once or be extended to a vector parameter, to control each signal's contribution individually.

up to 1, but can be treated independently. Dependent on the status of the gates, different edge cases can be parameterized:

- For $p_{ANN} = 0$ and $p_{FMU} = 1$, the system behaves like the original FMU without any ANN attached to it.
- For $p_{ANN} \neq 0$ and $p_{FMU} = 0$, the system behaves like a series connection of FMU and ANN.
- For $p_{ANN} \neq 0$ and $p_{FMU} \neq 0$, the system uses the FMU as well as the ANN dynamics. The ANN is able to manipulate the FMU dynamics (series) as well as learning its very own dynamics on given signals (parallel).

Finally, these gate parameters can be trained together with the ANN parameters and so bypass the initialization challenge of (physics-enhanced) NeuralODEs, which was further highlighted in (Thummerer, Stoljar, and Mikelsons 2022).

2.2.2 Data processing between FMUs and ANNs

FMUs and ANNs operate in different numerical ranges. Whereas the signal interface of FMUs is basically only limited by Float64, ANNs suffer if layer inputs become too large or too small dependent on the used activation functions. Small values from the FMU may not sufficiently influence the ANN output, large values lead to saturation if activation functions with limited output are used, like *tanh*, *sigmoid* or *relu*. A straight forward workaround for this is to scale and shift the FMU's output values to fit the activation function of the ANN, which is further described in (Thummerer, Stoljar, and Mikelsons 2022). To restore signal ranges at the ANN outputs, a post-processing transformation can be added here. If the ANN input signals are matching the outputs, this transformation can be initialized as inverse of the pre-processing transformation. The parameters of these pre- and post-processing operations can be initialized to suit known signal ranges and further be optimized together (as forward and reverse transformation) or separately (as individual transformations) with the ANN parameters to adapt well to changing signal ranges that might occur with further training convergence.

2.2.3 Signal selection

By design of the *Functional Mock-Up Interface* (FMI), different signals can be passed between FMU and ANN. Using all available FMU signals (in principle, any system variable can be retrieved by supported FMUs) results in suboptimal training performance, because unnecessary large Jacobian matrices need to be computed during training and the signal information is highly redundant, if besides system states and inputs further dependent variables are connected. Further, connecting signals that have no causal dependency on the physical effects being learned carries the risk of the ANN learning wrong correlations or at least delays training convergence. This motivates to define an efficient and minimal interface. A clever and automatic signal selection during training is an active topic of research. For now, choosing a small set of interface variables is driven by expert knowledge of the system and only some general instructions can be given. One is, that for higher order differential equations (mechanical system are usually modeled as second order ODEs), the order of differentiation of the ODE should be preserved (Thummerer, Stoljar, and Mikelsons 2022). This is achieved by preventing the ANN to modify the derivatives that are also states. For example, consider a translational mechanical system with the states *position* and *velocity* and the state derivatives *velocity* and *acceleration*. If the ANN is allowed to modify the entire derivative vector (here the velocity and the acceleration), the second order ODE disintegrates into a first order ODE, because the state *velocity* does not match the integrated *acceleration* anymore. This can be tackled by allowing the ANN to manipulate only the highest derivatives of the system, in the considered example the derivative *acceleration*, while passing

the derivative *velocity* directly to the numerical integrator without modifications.

2.3 From Julia to modeling environment

After training, the HM needs to be reimported from *Julia* back into the original (or another) development tool. For this, three different approaches to integrate the newly learned dynamic effect into the original modeling environment are possible and explained in the following sub sections.

2.3.1 Export the HM as FMU

The entire HM, including the FPM as FMU, can be exported as FMU, see Figure 6 (step 2a). This approach is useful if model development has (almost) finished and no other modifications to the structure needs to be done or if the HM is being used in a different modeling tool that features another modeling language (but support for FMI). From software side, this can be accomplished using the *Julia* library *FMI.jl* or the related sub-library *FMIExport.jl*[3] directly. An example on how to export NeuralFMUs as FMUs is part of the *FMIExport.jl* repository.

From a technical perspective, the FPM FMU (the FMU exported from the original simulation model) is copied to the resources folder of the HM FMU (the FMU being exported from *Julia*) and most of the FMI functions of the HM FMU are directly connected to the embedded FPM FMU. Because these connections are implemented by redirecting function pointers, the execution performance of the hybrid FMU is only influenced by the ANN dynamics without overhead. To induce the modified dynamics to the HM FMU, some functions need to be extended, which is discussed in the following paragraphs at the example of model-exchange FMUs.

Getting system state derivative
The functions `fmi2GetDerivatives` (FMI2) and `fmi3GetContinuousStateDerivatives` (FMI3) need to return the dynamics of the entire HM instead of the FPM FMU. This can be simply achieved by obtaining the FPM FMU state derivative and passing it through the part of the HM topology, that modifies the system state derivative. Finally, the modified state derivatives are returned instead of the derivatives of the FPM FMU.

Setting system state
The functions `fmi2SetContinuousStates` (FMI2) and `fmi3SetContinuousStates` (FMI3) set a new state for the FPM FMU. If the system state is manipulated by the HM topology before being passed to the FPM FMU, the HM FMU state is not matching the FPM FMU state. As a consequence, a new state for the FPM FMU needs to be retrieved by optimization through the topology (Thummerer, Stoljar, and Mikelsons 2022).

Getting/Setting ANN parameters
Before overwriting the corresponding functions, the model description needs to be extended by new parameter identifiers for the ANN parameters. After that, to be able to read and change the ANN parameterization, the functions `fmi2GetReal` and `fmi2SetReal` (FMI2) and `fmi3GetFloat64` and `fmi3SetFloat64` (FMI3) need to be overwritten to get and set corresponding ANN parameters.

Finally, further functions can be overwritten to add additional functionalities but are not highlighted at this point, for example function calls to `fmi2GetDirectionalDerivatives` (FMI2), `fmi3GetDirectionalDerivatives` and `fmi3GetAdjointDerivatives` (FMI3) can easily be chained to the *Julia* AD framework to efficiently retrieve partial derivatives.

2.3.2 Export the ANN only

Export as FMU
By exporting only the trained ANN as FMU, the FPM can further be used in its original format, maintaining the white-box structure that is known by the modeling engineer, see Figure 6 (step 2b). Exporting ANNs as FMUs is supported by *FMIExport.jl*, an example is part of the library repository.

Export in a dedicated format
Further, the ANN can be exported in a dedicated format instead of being compiled into an FMU, see Figure 6 (step 2c). Two different formats are discussed in the following.

From a linguistic point of view, the *Modelica* language is capable of describing any operation that is performed by an ANN layer. Even more uncommon layers, like the gates layer used in the presented use case, can be expressed with simple mathematical operations and therefore with *Modelica*. Currently, the ANN is exported as *Modelica* model by hand, but an automated export is part of an upcoming research project. The *Modelica* ANN can easily be imported into tools that support the *Modelica* language.

Further, also the *Open Neural Network Exchange*[4] (ONNX) can be used to export the ANN separately. ONNX is capable of describing conventional as well as custom layers, as long as they are subdivisible into primitive mathematical operations (like e.g. the gates layer). For *Julia*, an ONNX library is available, called *ONNXNaiveNASflux.jl*[5], that is able to import and export ONNX models including custom layers.

Instead of exporting the ANN structure alongside with the identified parameters, it is also possible to export both elements separately. This can be achieved by importing the ANN topology in the original modeling environment and transferring the parameters separately via an arbitrary file format (like `CSV`, `MAT` or `TXT`).

[3] `https://github.com/ThummeTo/FMIExport.jl` (accessed on May 24, 2023)

[4] `https://github.com/onnx/onnx` (accessed on May 30, 2023)

[5] `https://github.com/DrChainsaw/ONNXNaiveNASflux.jl` (accessed on May 31, 2023)

DOI 10.3384/ecp204141

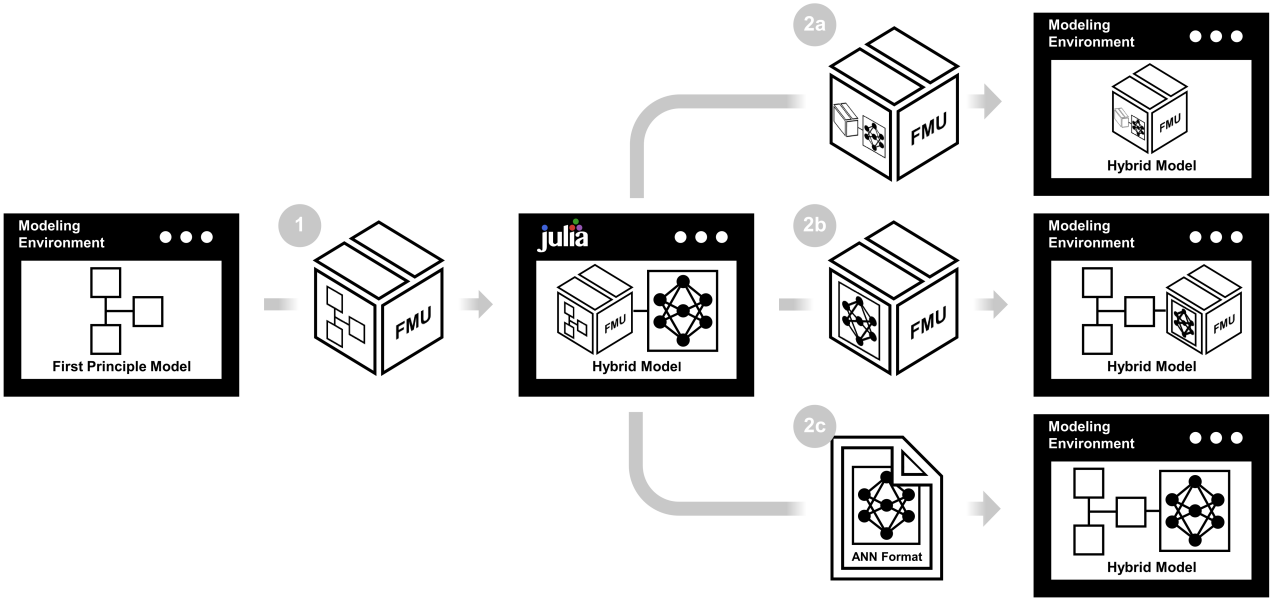

Figure 6. Comparison of different ways on how to deploy NeuralFMUs. Step (1) shows the export of the FPM as FMU from modeling environment and the import of the FMU into *Julia*. After hybrid modeling and training inside *Julia*, the entire HM can be exported as FMU (2a), only the ANN as FMU (2b) or a dedicated ANN format like *Modelica* or *ONNX* can be used for ANN export (2c). Dependent on the chosen exporting step, the further usability of the HM varies.

Reintegration of the exported ANN in the original modeling environment

The exported formats (FMU, ONNX, Modelica) share the input and output structure. When reintegrating them into the original modeling environment they all behave like data driven sub-models such as characteristic maps. The connection to the original FPM depends on the modeling tool's specifics. If the targeted modeling environment is a *Modelica* tool, the following has to be considered: The inclusion of the modified dynamics into the original FPM is not straight forward since the original *Modelica* system is acausal. The definition of variables as states or state derivatives only takes place during compilation. To modify a variable's derivative prior to compilation means to modify/replace equations in the *Modelica* system. This becomes a limitation, since ...

- ... component models could be protected,
- ... there are multiple equations to replace or modify, so the solution is not unique,
- ... sometimes variables/equations do not even exist (but are only created during compilation).

Hence, in general the integration of a causal ANN into the acausal *Modelica* model is not straightforward, and remains a future research topic.

3 Experiments

In the experimental part, the NeuralFMU approach is applied to a subsystem of an excavator model. Figure 7 sketches the excavator arm and the considered subsystem.

3.1 Motivation

The model of the excavator was developed in *SimulationX*[6] during the real excavator's setup period, in which the model was used to test an automatic steering controller. Now, during the operation of the excavator, the model shall be extended and used with measurement data as a *Hybrid Twin* to detect and predict faults (i.e. deviations, leaks in hydraulic components, etc.) (Gundermann et al. 2018). The *Hybrid Twin* concept enhances physics-based models with real data from physical sensors to increase the model's accuracy and prediction capabilities (Chinesta et al. 2019). The advantage of the applied *Hybrid Twin* of the excavator - a condition monitoring system of the latter - is that the real system requires less scheduled maintenance while at the same time can be repaired in time before serious faults occur. One major prerequisite for detecting deviations using *Hybrid Twins* is that the simulated nominal behavior must match the measured data very accurately during operation. This can be acchieved by applying the NeuralFMU method to the system. The results for the bucket cylinder subsystem of the excavator (including reimport to the original model) are outlined below.

3.2 Model selection for NeuralFMU

The complete, comprehensive model of the excavator contains components of various domains of the *SimulationX* libraries, such as *Multi Body Systems* (MBS, i.e. 3D) and 1D mechanics, hydraulics and control signals. The real excavator is equipped with pressure and cylinder position

[6] `www.SimulationX.com` (accessed on May 24, 2023)

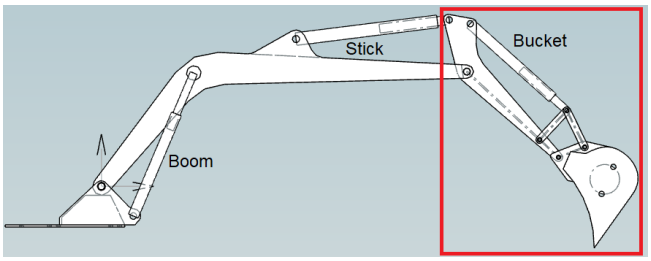

Figure 7. Sketch of the full excavator arm with all three cylinders. For the NeuralFMU approach, the subsystem controlled by the bucket cylinder (red box) is used.

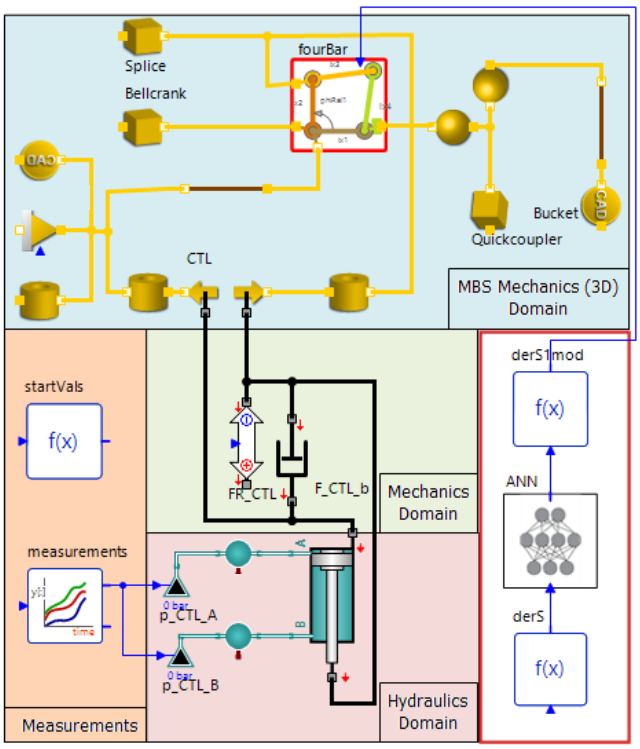

Figure 8. The submodel of the bucket cylinder and mechanics in *SimulationX*. The model shows different domains (color marked): MBS mechanics (blue), 1D mechanics (green), hydraulics (red), signal blocks for measured pressures and initial position values (orange). The original FPM is shaded with the colors of the domains. The extension of the model to a Hybrid Model (cf. subsection 3.4) is highlighted by the red boxes.

sensors, the signals of which can be recorded together with the control signals. This allows to collect data of the motion of a single cylinder as well as of the full motion of the excavator arm. Comparing model with data, one will find that there are deviations, e.g. between simulated and measured cylinder positions.

One source of deviation is the friction in the hydraulic cylinders, which is hard to model. Simple models are not able to describe the process accurately enough, sophisticated models are hard to parameterize, and the modeling task is even more complex since different parts in the cylinder system contribute to the effective friction force. To avoid the influence of interaction with connected components, the model is considered for each cylinder separately. Therefore, the bucket cylinder subsystem and attached mechanics are cut out of the full model and are used with the measured pressure signals in the cylinder chambers as inputs, to focus on the movement of the considered cylinder only. Figure 8 shows the submodel of the bucket cylinder and mechanics, which will be used in experiments, and be referred to as the *First Principle Model* (FPM). The model already includes some generic friction components - a pressure based force `FR_CTL` and a damper `F_CTL_b`, which prevents oscillations. The necessity of including or excluding such components will be examined below.

It shall be mentioned that similar submodels can be created for the other cylinders of the excavator arm, in which case there are more components on the lower end of the kinematic tree which affect the mass and momentum distribution. Note, that the system model is unstable if the center of mass of the rotated components lies above the joint around which the mechanical components rotate when extending or retracting the cylinder. This can lead to difficulties, e.g. when trying to fit the friction force for the boom cylinder. Even with submodels in a stable state, the fitting method is time-consuming and requires knowledge of engineers with system specific experience.

The cylinder chambers in Figure 8 are fed with non-constant pressure signals, taken from real measurements shown in Figure 9. These pressure signals cause the piston in the cylinder to move. As mentioned, the simulated position deviates from the measurement, which can be seen in Figure 14 which spans the same time sequence. The position of the bucket cylinder will be used as training objective in the NeuralFMU optimization. Further, the measurement data is noisy and contains irregular deviations (e.g. position measurement around $470\,s$), which cannot be removed by standard filters.

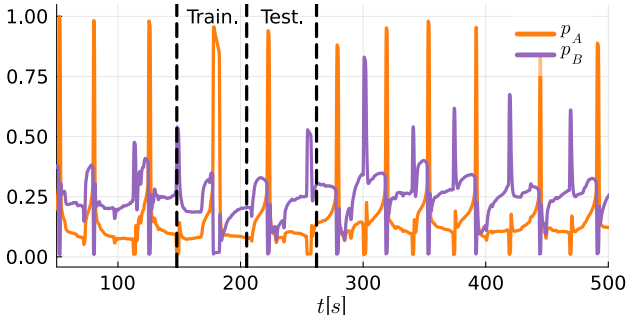

Figure 9. Measured pressure signals p_A and p_B of the two cylinder chambers (normalized, units removed).

3.3 NeuralFMU setup and validation

3.3.1 The FMU of the Bucket model

For applying the NeuralFMU method, an FMU of the FPM is generated, with the top bucket cylinder position `CTL.dx` as the output y. Figure 10 shows two extraction-retraction-cycles of the time segment used to train and test the NeuralFMU. For the NeuralFMU training, a single

DOI
10.3384/ecp204141

extraction-retraction-cycle ranging from 148 s to 205 s is selected from the data since it covers most of the possible dynamics of the measured series. A second cycle ranging from 205 s to 262 s is used for testing. Please note, that the amount of data for training is very small in terms of machine learning applications and should be increased to contain every aspect of the effect to be learned. Here however, it is shown that even small and incomplete data sets can be used to significantly improve the simulation model.

Table 1. FMU states $\boldsymbol{x}$ and output y.

Symbol	Name	Description
x_1	`der(fourBar.q[2])`	Angular velocity
x_2	`connection12.x`	Position
x_3	`fourBar.q[2]`	Angle
y	`CTL.dx`	Position

As shown in Table 1, the FMU has three states, which seems not intuitive, since the mechanical system (cf. Figure 8) has only one degree of freedom: the motion around the revolute joint in the four bar. The related states are the angle `fourBar.q[2]` and the corresponding angular velocity `der(fourBar.q[2])`. The third state `connection12.x` is the position of the 1D mechanical connection between force interface (`CTL`) and hydraulic cylinder port B, which is equivalent to the length of the bucket cylinder. This state is introduced since the 3D-1D force interface creates a constraint on the 1D velocity. Hence a differential equation for the 1D position (`connection12.x`) exists. Further, the bucket cylinder position `CTL.dx` is identical to the state `connection12.x` in the considered simulation range. Because of the zero-crossing behavior in `der(fourBar.q[2])`, the FMU has state events that must be handled during simulation to obtain correct simulation results (Blochwitz et al. 2011).

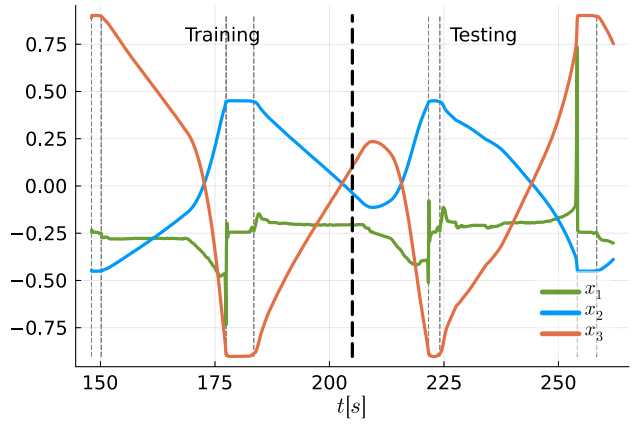

Figure 10. FMU states $\boldsymbol{x}$ (normalized, units removed) during training and testing. State events (discontinuities) are triggered whenever the cylinder reaches or leaves one of the end stops. In the graph, they are shown as gray-dashed horizontal lines.

3.3.2 The definition of the ANN

The definition of the ANN topology bases on past experience and empirical hyper parameter tuning, and is considered as large enough to cover the dynamics, while being trainable in reasonable time. In addition to a core of two dense layers, pre- and post-processing layers are included, as shown in Figure 11. These shift and scale the value of all inputs to a distribution with mean 0 and standard deviation 1 to suit the applied activation function *tanh* of the first dense layer. The figure further shows the additional gates layer introduced in subsubsection 2.2.1 which allows efficient training of the unknown friction effect and a solvable initialization for the system. The final ANN topology with 91 parameters is shown in Table 2.

Table 2. Parameters of the NeuralFMU topology.

Type	Inputs	Outputs	Bias	Num.
Pre-process	3	3	0	6
Dense	3	16	16	64
Dense	16	1	1	17
Post-process	1	1	0	2
Gates	2	1	0	2
			Total:	91

3.3.3 Solver details and loss function

To solve the NeuralFMU in *FMIFlux.jl*, the explicit Runge-Kutta method *Tsit5* (Tsitouras 2011) is used as numerical ODE solver. Using the gradient-based optimization algorithm *Adam* (Kingma and Ba 2017) with step size 10^{-3}, the following loss function of the mean absolute error (*mae*) is minimized during training:

$$mae(\boldsymbol{x}_2, \hat{\boldsymbol{x}}_2) = \frac{1}{n}\sum_{i=1}^{n}\left|x_2^i - \hat{x}_2^i\right|, \tag{1}$$

where x_2^i is the simulated bucket cylinder position `connection12.x` at time instant i, $\hat{x}_2^i$ the corresponding measured position, and n the number of compared measurement points.

3.3.4 Training and testing in Julia

After defining the ANN topology and its training parameters, the NeuralFMU is trained in *Julia* for 2000 epochs on the first extraction-retraction-cycle of the selected time interval ranging from 148 s to 205 s in 0.1 s second increments. Two experiments are considered: The training of a *Hybrid Model* (HM) on basis of the FPM with a simple friction model and on basis of the FPM without friction (undamped system).

FPM with friction model

The results after training of the NeuralFMU are very close to the measurement data, which is confirmed by the small value of the loss function from Equation 1 of 0.0094 m. After training is complete, the NeuralFMU is simulated the second extraction-retraction-cycle in the time interval

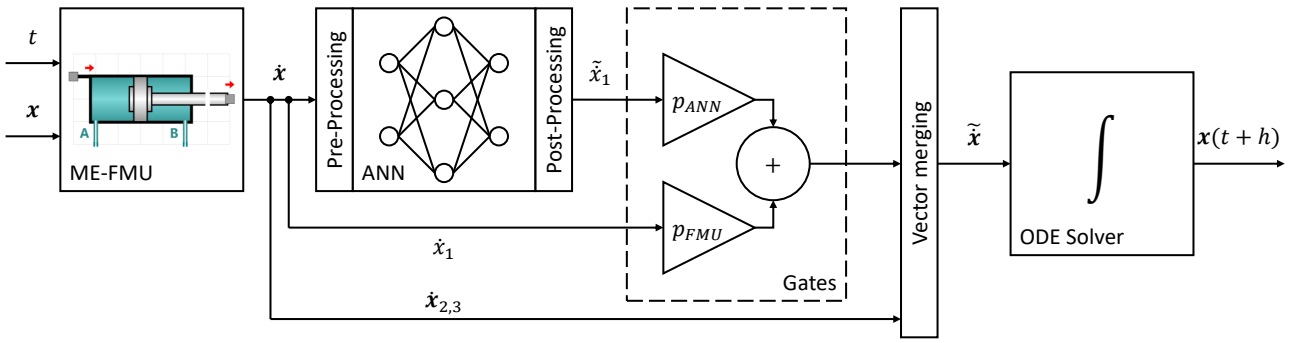

Figure 11. The topology of the used NeuralFMU using gates adapted from (Thummerer, Stoljar, and Mikelsons 2022). The current system state $\boldsymbol{x}$ and time t are passed to the FMU, which computes the system state derivative $\dot{\boldsymbol{x}}$. On basis of the state derivatives from FMU, only the revolute joint acceleration $\tilde{\dot{x}}_1$ is computed by the ANN (featuring pre- and post-processing layers) and linearly combined with the corresponding derivative from the FMU $\dot{x}_1$ in the gates layer. The remaining FMU derivatives $\dot{\boldsymbol{x}}_{2,3}$ only enter the ANN as inputs and are bypassed directly to the final derivative vector $\tilde{\dot{\boldsymbol{x}}}$, to preserve a second order ODE.

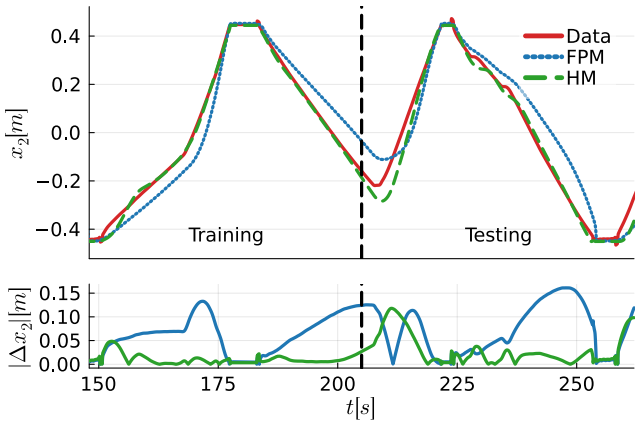

Figure 12. Comparison of bucket cylinder position to data (top) and absolute deviation from data (bottom). The HM is modeled on basis of the FPM with simple friction.

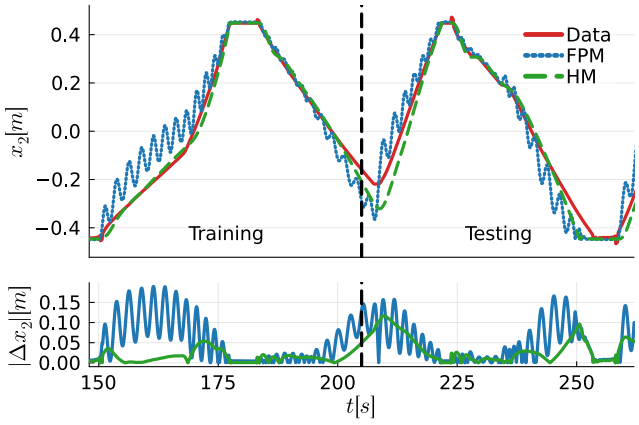

Figure 13. Comparison of bucket cylinder position to data (top) and absolute deviation from data (bottom). The HM is modeled on basis of the FPM without friction.

from 205 s to 262 s, shown in Figure 12. The test result shows that the loss function value of 0.0228 m doubled compared to the training sequence, but it is still three times smaller than the loss function value of 0.0696 m in the original FPM FMU simulation (cf. in Table 3).

FPM without friction model

As introduced, another NeuralFMU is built using an FPM without friction components. Training and testing results for the same sequence are shown in Figure 13. These confirm the fact, that the NeuralFMU can be trained completely without friction components and the ANN can compensate the oscillations of the border-stable system. The loss function value of the NeuralFMU featuring the FPM without friction is 0.0327 m and therefore about 30 % higher than the one trained on basis of the FPM with friction for the same sequence (cf. in Table 3). The accuracy of the case without friction under the same training conditions is lower than the one implementing viscous damping, therefore the model with friction is used for the outlined application.

Table 3. Loss function values.

Model	FPM friction	Training [m]	Testing [m]
FPM	yes	0.0616	0.0696
HM	yes	0.0094	0.0228
FPM	no	0.0543	0.0497
HM	no	0.0157	0.0327

3.4 Hybrid Model in SimulationX

As described in section 2.3.2, one way to reimport the NeuralFMU to the original modeling environment is to export the ANN in a dedicated format and to couple it with the FPM (or an FMU containing the FPM) inside the modeling tool. Thereby after training and testing in *Julia*, the ANN parameters are exported as a `TXT` file and a *Modelica* model with an equivalent network topology (cf. Figure 11) and the parameter values loaded from this text file is created. Listing 1 shows the equation section of the ANN type implemented in *SimulationX*.

Listing 1. Equation section of Modelica type of ANN

```
preProcess  = (dxIn+prePShift).*prePScale;
```


DOI
10.3384/ecp204141

```
dense1      = tanh(preProcess*w1 + b1);
dense2      = dense1*w2 + b2;
postProcess = dense2*postPScale+postPShift;
dxOut={
   gates[1]*dxIn[1]+ gates[2]*postProcess,
   dxIn[2],
   dxIn[3]};
```

Here, `dxIn` and `dxOut` are the state derivatives as calculated by the FPM and modified by the ANN, respectively. `prePShift`, `prePScale`, `w1`, `b1`, `w2`, `b2`, `postPScale`, `postPShift` and `gates` are the optimized parameters of the different layers. To include the ANN in the FPM, equations have to be modified (cf. Sec. 2.3.2). In the bucket system (see Figure 8), this affects only the equation which defines the angular acceleration in the four bar. The variables defining the FMU state derivatives (cf. Table 1) are read from the model components (`derS`), and are passed as input into the `ANN`. The modified `der(fourBar.q[2])` enters the `FourBar` component as an input (named `derS1mod`) and is used instead of the original variable. The equation section in the four bar is modified as written below:

Listing 2. Modification of FourBar equation section

```
der(q[2]) = om[2];      //-> input 3 to ANN
alp[2] = ...            //-> input 1 to ANN
//der(om[2]) = alp[2];  replaced by:
der(om[2]) = derS1mod; //<- output 1 of ANN
```

In Figure 8, the extension and modifications of FPM by the ANN are highlighted with red boxes. The bottom right box shows state derivatives (`derS`), `ANN`, and modified state derivative of state x_1 (`derS1mod`). This modified derivative is fed into the modified `fourBar` component (also highlighted with red box).

The output of the HM with a simple friction model simulated in *SimulationX* in the interval from 50 s to 500 s is shown in Figure 14. The figure also shows that the bucket cylinder position can be reliably predicted in *SimulationX* for a measurement sequence that was not used to train and test the NeuralFMU. Comparing the results in the time interval for training and testing, the NeuralFMU shows the same cylinder position in *Julia* as well as in *SimulationX* for the original FPM and HM without friction, as can be seen when comparing results in *SimulationX* (Figure 14 and 15) with results in *Julia* (Figure 12 and 13). Figure 15

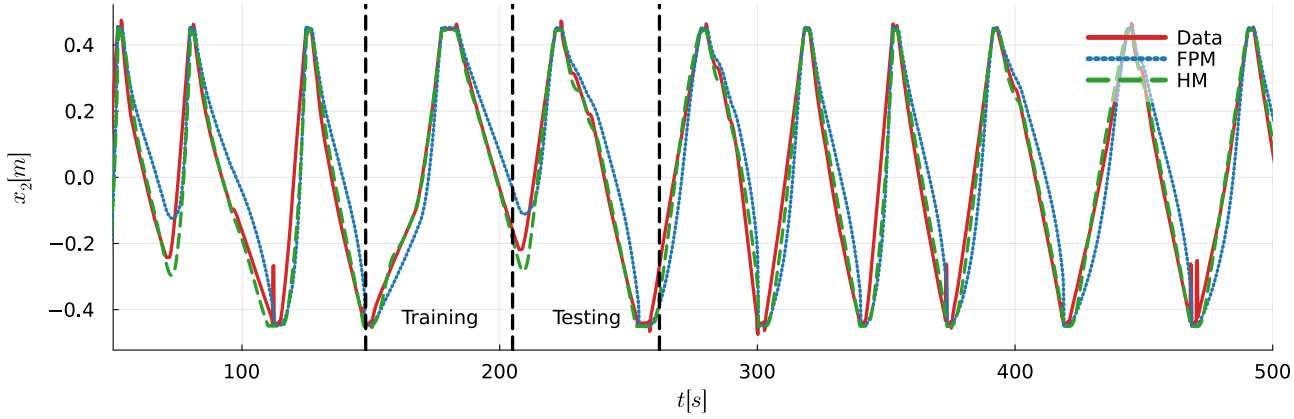

Figure 14. Measured data and simulation results of FPM and HM for the bucket cylinder position for the entire measurement. The HM is modeled on basis of the FPM with simple friction.

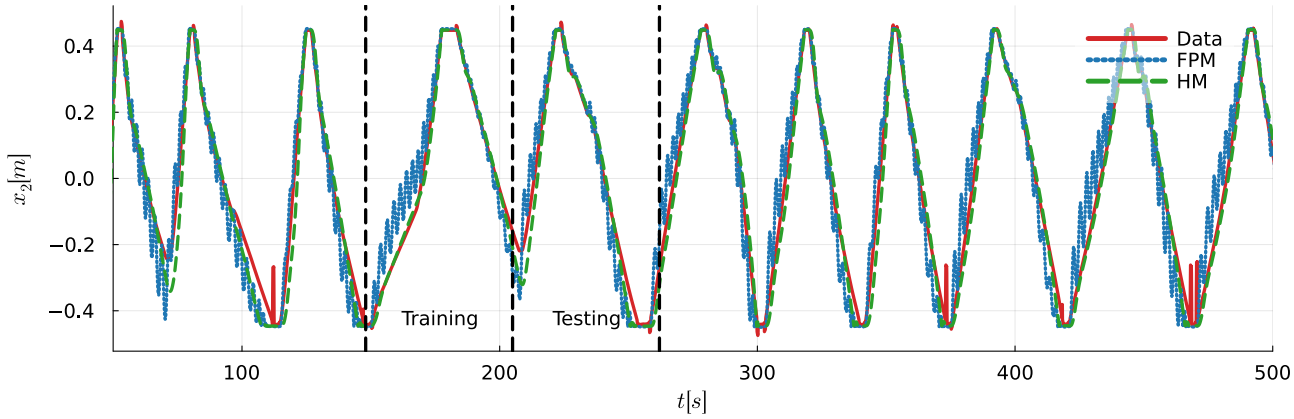

Figure 15. Measured data and simulation results FPM and HM for the bucket cylinder position, for the entire measurement. The HM is modeled on basis of the FPM without friction.

also confirms that the HM without friction is able to damp oscillations for the full measurement sequence.

4 Conclusion

We started by introducing the concept of a NeuralODE and adapted this machine learning model step by step to suit the requirements of industrial engineering applications, resulting in a so called NeuralFMU. We highlighted a generic workflow, that allows for NeuralFMU development in custom applications, dependent on tool capabilities and further model use. Finally, we exemplified the presented theory at a real engineering use-case: The modeling of a *Hybrid Twin* of a hydraulic bucket cylinder to be used for process and failure monitoring.

Next, additional features of the *FMIFlux.jl* package shall be investigated - input values to NeuralFMUs for interaction with other excavator components and the batching method for more effective training on a broader data base. Besides, we plan to apply the workflow to larger submodels and other examples, and to automate parts of the workflow where possible. For the excavator model, the final goal is the creation of the *Hybrid Twin* for the detection of malfuntions in the system.

Besides physical equations, further system knowledge in the form of ODE properties like stability, oscillation capability, stiffness as well as frequency and damping information can be integrated into the PeNODE training process in form of Eigen-informed NeuralODEs (Thummerer and Mikelsons 2023). All listed properties depend on the eigenvalue positions of the system model over time. By computing eigenvalues and rating their positions as part of the (or an additional) loss function, the desired ODE properties can be enforced for the considered *Hybrid Twin* and may further improve training and prediction.

Author Contributions

NeuralFMU method, implementation and support: T.T., L.M.; Experimental results in *Julia* and *SimulationX*: A.K, J.G.; Preparation of excavator model and data: D.R.; Writing: A.K, T.T., J.G., L.M.; All authors have read and agreed to the published version of the manuscript.

Funding

The contributions to this research were partially funded by different projects:

- the ITEA3-Project Unleash Potentials in Simulation (UPSIM) Nr. 19006. `https://www.upsim-project.eu/`
- Bauen 4.0, funded by German Federal Ministry of Education and Research (grant number 02P17D230). `https://www.verbundprojekt-bauen40.de/`
- Proper Hybrid Models for Smarter Vehicles (PHyMoS), funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) (grant number 19I2022A). `https://phymos.de/` (all accessed on May 23, 2023)

The authors thank the organizations for their funding.

References

Bezanson, Jeff et al. (2014). "Julia: A Fresh Approach to Numerical Computing". In: *CoRR* abs/1411.1607. arXiv: 1411.1607. URL: http://arxiv.org/abs/1411.1607.

Bittner, L. (1963). "L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, E. F. Mishechenko, The Mathematical Theory of Optimal Processes. VIII + 360 S. New York/London 1962. John Wiley & Sons. Preis 90/–". In: *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 43.10-11, pp. 514–515. DOI: 10.1002/zamm.19630431023.

Blochwitz, T. et al. (2011). *The functional mockup interface for tool independent exchange of simulation models*. URL: https://publica.fraunhofer.de/handle/publica/371243.

Champaney, Victor et al. (2022). "Modeling systems from partial observations". In: *Frontiers in Materials* 9. ISSN: 2296-8016. DOI: 10.3389/fmats.2022.970970.

Chen, Tian Qi et al. (2018). "Neural Ordinary Differential Equations". In: *CoRR* abs/1806.07366. arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366.

Chinesta, Francisco et al. (2019). "Virtual, Digital and Hybrid Twins: A New Paradigm in Data-Based Engineering and Engineered Data". In: *Archives of Computational Methods in Engineering*. DOI: 10.1007/s11831-018-9301-4.

Gundermann, Julia et al. (2018). "The Fault library - A new Modelica library allows for the systematic simulation of non-nominal system behavior". In: *Proceedings of the 2nd Japanese Modelica Conference*. Linköping Electronic Conference Proceedings 148. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, pp. 161–168. DOI: 10.3384/ecp18148161.

Kingma, Diederik P. and Jimmy Ba (2017). *Adam: A Method for Stochastic Optimization*. arXiv: 1412.6980 `[cs.LG]`.

Thummerer, Tobias, Josef Kircher, and Lars Mikelsons (2021-09). "NeuralFMU: Towards Structural Integration of FMUs into Neural Networks". In: *Proceedings of the 14th International Modelica Conference*. Ed. by Martin Sjölund et al. Linköping Electronic Conference Proceedings 181. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, pp. 297–306. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181297.

Thummerer, Tobias and Lars Mikelsons (2023). *Eigen-informed NeuralODEs: Dealing with stability and convergence issues of NeuralODEs*. arXiv: 2302.10892 `[cs.LG]`.

Thummerer, Tobias, Johannes Stoljar, and Lars Mikelsons (2022). "NeuralFMU: Presenting a Workflow for Integrating Hybrid NeuralODEs into Real-World Applications". In: *Electronics* 11.19. ISSN: 2079-9292. DOI: 10.3390/electronics11193202.

Thummerer, Tobias, Johannes Tintenherr, and Lars Mikelsons (2021-11). "Hybrid modeling of the human cardiovascular system using NeuralFMUs". In: *Journal of Physics: Conference Series* 2090.1, p. 012155. DOI: 10.1088/1742-6596/2090/1/012155.

Tsitouras, Ch. (2011). "Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption". In: *Computers & Mathematics with Applications* 62.2, pp. 770–775. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2011.06.002.