Accelerating the simulation of equation-based models by replacing non-linear algebraic loops with error-controlled machine learning surrogates

Andreas Heuermann [1] Philip Hannebohm [1] Matthias Schäfer[2] Bernhard Bachmann [1]

[1]Institute for Data Science Solutions, Bielefeld University of Applied Sciences and Arts, Germany,
{first.last}@hsbi.de
[2]LTX Simulation GmbH, Germany, Matthias.Schaefer@ltx.de

Abstract

When simulating a Modelica model, non-linear algebraic loops may be present, which involves solving multiple equations simultaneously. The classical Newton-Raphson method is commonly employed for solving a non-linear equation system (NLS). However, the computational burden of using this method during simulation can be significant. To tackle this issue, utilizing artificial neural networks (ANNs) to approximate the solution of algebraic loops is a promising approach. While ANN surrogates offer fast performance, ensuring the correctness of the computed solution or quantifying reliability can be challenging.

This publication presents a prototype, based on the OpenModelica compiler (OMC) (Fritzson et al. 2020), that automates the extraction of time-consuming algebraic loops. It generates training data, trains ANNs using machine learning (ML) methods, and replaces the algebraic loops with ANN surrogates in the simulation code. A hybrid approach, combining the trained surrogate with the nonlinear Newton solver, is then used to compute the solution with a desired level of accuracy.

Keywords: Machine Learning, Dynamic Systems, Surrogate Model, Non-Linear System, Error Control

1 Introduction

Modelling and simulation play a major role in many fields of science, technology, engineering and mathematics. Modelica (Mattsson and Elmqvist 1997) is an established object-oriented language for multi-domain modeling. It is easy to develop model-based components using simple textbook equations and combine them into detailed and complex cyber-physical systems. With increasing complexity even on modern Modelica compilers simulation performance can slow down.

One way to computational accelerate Modelica components is using ML surrogates. Such a surrogate approximates the equation-based Modelica model with a data-driven approach. When sufficiently trained, a surrogate can replace the corresponding Modelica equations and the resulting speedup can be utilized e.g., in parameter optimization.

Different data-driven ML methods are used in the context of modelling and simulation. ANNs as methods of artificial intelligence (AI) are often used, in particular physics informed neural networks (PINNs) (Lawal et al. 2022), long short-term memory (LSTM) networks (Hochreiter and Schmidhuber 1997), continuous-time echo state networks (CTESNs) (Anantharaman et al. 2020) could demonstrate impressive speedups of simulation time for complex models. While these methods are fast and precise no guarantees for correctness can be made that the surrogate solutions stays within the desired error tolerances.

So called hybrid physical-AI based models are a compromise between classical and ML models. A hybrid model can consist of equations derived from first principle physics as well as data-driven ML models. They offer better simulation performance with acceptable accuracy. While (Hübel et al. 2022) could show improved performance with a reduced order model the resulting hybrid model cannot be used outside of the trained area or ensure a given error tolerance.

In this publication the authors present a partially automated method to replace non-linear algebraic loops of Modelica models with error-controlled ML surrogates to generate hybrid physical-AI based models. The relation between inputs and outputs of the loop are learned from synthesized data and reference simulations. It could be shown, that with the use of ANN the simulation time could be sped up by a factor of 1.5 while keeping the surrogate prediction within a given error tolerance.

This enables users to select the tradeoff between accuracy and speedup.

Paper Organization

Subsection 3.1 describes the use of profiling to identify NLSs that are worthwhile to replace. Different methods to generate artificial training data from the original model are discussed in Section 3.2. The exemplary training of feedforward neural networks (FNNs) is illustrated in Section 3.3 while Section 3.4 shows an approach to reduce the demand for generated training data. Section 3.5 presents

the integration of the trained surrogate into the simulation while the error control is discussed in Section 3.6. Results are shown in Section 4.2.2 with models from the Scalable-TranslationStatistics Modelica library. Finally section 5 discusses results and encountered problems and section 6 concludes the paper.

2 Problem Statement

Developing a detailed high fidelity Modelica model named M_{hf} is an elaborated task and simulating such a model can take a significant amount of time. For applications like parameter fitting simulation speed outweighs fidelity, so another Modelica model named M_{sur} is needed to complete the given task in an acceptable amount of time. The classic approach is to manually replace expensive equations of the original M_{hf} model and reduce the complexity of the modeled physics. Another approach is to generate large sets of artificial data from the expensive to solve M_{hf} model to then train a ML surrogate.

The ordinary differential equation (ODE) of M_{hf} can have subsystems of equations that need to be evaluated simultaneously. These subsystems, also known as algebraic loops, can consist of linear or non-linear equations. This paper only discusses the treatment of non-linear loops, since in general they are harder to solve than linear loops.

An algebraic loop can be described in its residual form

$$\begin{aligned} f_{res} &: I_t \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_{in}} \times \mathbb{R}^{n_{out}} \to \mathbb{R}^{n_{out}}, \\ & f_{res}(t, p, z_{in}, z_{out}) \overset{!}{=} 0 \end{aligned} \tag{1}$$

with simulation time $I_t := [t_{start}, t_{stop}] \subset \mathbb{R}$, parameters $p \in \mathbb{R}^{n_p}$, used variables $z_{in} \in \mathbb{R}^{n_{in}}$ computed in preceding model equations and unknowns $z_{out} \in \mathbb{R}^{n_{out}}$. Define a function

$$f_{NLS} : I_t \times \mathbb{R}^{n_p} \times \mathbb{R}^{n_{in}} \to \mathbb{R}^{n_{out}}, \;\; f_{NLS}(t, p, z_{in}) = z_{out} \tag{2}$$

that solves Equation 1 explicitly. For simplicity non-unique solutions to Equation 2 are ignored for now. In practice this function is approximated by iterative root finding methods solving Equation 1, for example the Newton-Raphson method.

Instead of replacing all equations of M_{hf} with ML surrogates the authors propose to replace only the slowest non-linear equation systems f_{NLS} with faster ML surrogates f_S to reduce the time each evaluation of the right-hand side of the ODE takes.

Because the residual f_{res} and its Jacobian J are available, it is possible to compute error approximations for prediction $\tilde{z}_{out} = f_S(t, p, z_{in})$. Therefore it is possible to ensure, that the prediction $\tilde{z}_{out}$ stays within a given tolerance and the default non-linear solver can improve the solution if necessary.

By using an error controlled surrogate overall simulation time could be improved by a factor of 1.5 as shown in Section 4.2.2.

3 Method

Dependency information of the equations as well as a callable C function are necessary to replace f_{NLS} with a fast surrogate f_S. Because of this the authors choose the open-source OMC. It offers ways to interfere with the compilation process in order to retrieve the aforementioned requirements.

The method to generate a FMU containing the fast surrogates for slow non-linear equation systems consists of four main steps:

1. Find NLSs worth replacing (Section 3.1).
2. Generate training data (Section 3.2).
3. Train surrogates (Section 3.3).
4. Integrate trained surrogates into original model (Section 3.5).

Steps 1, 2 and 4 can be handled by the prototype implementation while the training process in step 3 still needs human intervention.

3.1 Profiling Model

The first step consists of analyzing how much time is spent for each NLS in relation to the total simulation time. Profiling for each model equation is performed to find all NLSs that need more of the total simulation time than a defined threshold. Since NLSs from the initial systems are solved only once at simulation time $t = t_{start}$ they are not considered for replacement.

Solving an NLS is time consuming, independent of the chosen Modelica tool. The example `ScaledNLEquations.NLEquations_5` from Section 4.2 was profiled in two popular tools: Dymola and OpenModelica. Most of the simulation time is spent solving the 8 non-linear systems, as can be seen in Table 1. Dymola spends around 53.05% and OpenModelica spends around 51.50% of the total simulation time solving these non-linear systems.

To find suitable equations for surrogate replacement the OpenModelica profiler (Sjölund 2015) is used to identify the Modelica equations corresponding to the slowest NLSs. The generated simulation results are used in Section 3.2 to specify the relevant input space for generating training data.

3.2 Data Generation

After identifying equations for replacement data driven surrogates need training, validation, and test data.

For the NLS all reached values for z_{in} are recorded. To only generate data in the area of interest the reference solution from the profiling step can be used to define a hypercube around the reference variables:

$$H_{in} := I_t \times I_{in} \tag{3}$$

$$I_{in} := \{ z \in \mathbb{R}^{n_{in}} \mid a_j \le z_j \le b_j \forall j \} \tag{4}$$

where

$$a_j := \min_{t \in I_t} z_{in}(t)_j, \; b_j := \max_{t \in I_t} z_{in}(t)_j. \tag{5}$$

 DOI 10.3384/ecp204275

Table 1. Profiling non-linear equation systems of ScalableTranslationStatistics.Examples.ScaledNLEquations.NLEquations_5

OpenModelica			*Dymola*		
Index	*Total* [s]	*Fraction* [%]	*Block*	*Total* [s]	*Fraction* [%]
	5.892	100.00		4.830	100.00
907	0.434	7.54	1100	0.334	6.91
928	0.433	7.51	1117	0.334	6.91
834	0.423	7.35	1187	0.333	6.91
813	0.396	6.88	1136	0.327	6.76
882	0.392	6.81	1204	0.323	6.68
951	0.392	6.80	1153	0.322	6.65
857	0.386	6.70	1170	0.319	6.60
987	0.111	1.92	1083	0.272	5.63

If the user has knowledge of the input variables it is viable to refine I_{in} to limit the training area further. For example physical constraints or boundary conditions can enforce that the surrogate only has to be valid in a specific region or that some combinations of different inputs are not reachable.

For the training process the input space I_{in} must be sampled sufficiently dense and corresponding solutions saved. One straightforward approach is to initialize the model at time t_{start} to set all constants and parameters. Subsequently pairs (z_{in}, z_{out}) are computed, where $z_{in} \in I_{in}$ random and z_{out} is computed by Equation 2.

Improvements of the data generation regarding the ANN training effort are discussed in Section 3.2.4.

3.2.1 `evaluateEquation` C Interface

All equations f_{NLS} can be evaluated with `evaluateEquation` without evaluating any other equations. The input variables z_{in} must be set using the appropriate `set` function before evaluating the equation. Afterwards the solution z_{out} can be inquired with the corresponding `get` functions.

```
status evaluateEquation(model c,
                        const size_t eqNumber);
```

- Argument `c` is the pointer to the model specific data structure of OMC.
- Argument `eqNumber` specifies the unique equation index of the equation to be evaluated.
- If the equation was successfully evaluated `success` is returned, otherwise `discard` is returned.

3.2.2 Implementation Details

The methods described in this paper are implemented in the prototypical Julia package NonLinearSystemNeuralNetworkFMU.jl[1]. For the function `evaluateEquation` an OpenModelica source-code ModelExchange 2.0.4 Functional Mock-up Unit (FMU) is built from the original Modelica model `M` using the following compiler flags[2]:

```
--fmiFilter=internal
--fmuCMakeBuild=true
--fmuRuntimeDepends=modelica
```

Then the C source files for `evaluateEquation` are included into the FMU. The binaries and `M.fmu` are recompiled using the provided `CMakeLists.txt` file. The resulting extended FMU is called `M.interface.fmu`.

To compute input-output pairs (z_{in}, z_{out}) Julia package FMI.jl[3] (Thummerer, Mikelsons, and Kircher 2021) is used to instantiate the FMU, set z_{in} and call `evaluateEquation` to evaluate $f_{NLS}(t, p, z_{in})$ with the Newton-Raphson method. The resulting input-output pairs (z_{in}, z_{out}) are saved to a CSV file for each NLS.

The Functional Mock-up Interface (FMI) standard is used because it provides a standardized way to instantiate, initialize, and solve an ODE system. At the time of writing FMI.jl allows the illegal call of `fmi2SetXXX`, allowing this workflow for a provisional but functioning prototype. This might change in a future version. In a matured implementation these steps must be performed directly in the Modelica compiler or its simulation runtime.

3.2.3 Preprocessing Data

It is possible for a NLS to have multiple solutions $z_{out_1} \neq z_{out_2}$, such that for the same input z_{in}

$$f_{res}(t, p, z_{in}, z_{out_1}) = f_{res}(t, p, z_{in}, z_{out_2}) = 0. \quad (6)$$

In this case Equation 2 is multi-valued and no unique functional relation exists. In case the training data has two sets

[1] NonLinearSystemNeuralNetworkFMU.jl v0.5.1: github.com/AnHeuermann/NonLinearSystemNeuralNetworkFMU.jl

[2] OpenModelica User's Guide on compiler flags: openmodelica.org/doc/OpenModelicaUsersGuide/1.20/omchelptext.html

[3] ThummeTo/FMI.jl v0.10.2: github.com/ThummeTo/FMI.jl

of input data that are close together but the corresponding outputs are far apart, the surrogate will not be able to approximate it. The training data has to be processed in a way that the relation from input to output is sufficiently continuous, i.e. not jumping between different solution branches.

Modelica tools usually follow one solution continuously if the step size of the ODE solver is small enough and the previous solution of the NLS is a good enough start value for the root finding method. Algorithm 1 from Section 3.2.4 tries to mimic this behavior, so on one trajectory the solution should not jump between different branches. However this only guarantees local uniqueness as two different trajectories to an input z_{in} can still lead to two different outputs $z_{out_1} \neq z_{out_2}$.

3.2.4 Improving Data Generation

Iterative non-linear solver methods need a decent start value to converge to a solution. Whether the method converges and if so at which rate highly depend on the chosen start values. An intuitive method to generate training data using small random perturbations is described in Algorithm 1.

Algorithm 1 Random Walk

1: **procedure** RANDOMWALK(δ, Δ_t)
2: $z_{out} \leftarrow 0$
3: $z_{in} \leftarrow$ random value from I_{in}
4: **for** $t = t_{start}, t_{start} + \Delta_t, \ldots, t_{end}$ **do**
5: $z_{out} \leftarrow f_{NLS}(t, p, z_{in})$, using previous z_{out} as start value
6: Save (z_{in}, z_{out})
7: $z_{in} \leftarrow z_{in} + \delta\omega$, where $\omega \in [-1, 1]^{n_{in}}$ random
8: Ensure $z_{in} \in I_{in}$

With small enough $\delta > 0$ and $\Delta_t > 0$ the number of iteration steps needed to solve the NLS for a given tolerance should be low (close or equal to 1), since the previously computed solution z_{out} is a good start value for the next small random perturbation of input vector $z_{in} \leftarrow z_{in} + \delta\omega$. The data generation process consists of creating several of these `randomWalk` trajectories to cover I_{in} densely.

3.3 Supervised Learning

Using the generated training data from Section 3.2 it is possible to train a ML surrogate for each NLS. This paper restricts itself to simple FNN models:

$$model_1(t, z_{in}) \approx f_{NLS}(t, p, z_{in}) = z_{out} \tag{7}$$

Due to implementation limitations parameters p are not changeable now and constant during the training process. That means each parameter configuration requires its own surrogate. It is planned to address this in future work.

To solve the issue of ambiguous solutions a different FNN

$$model_2(t, z_{in}, \hat{z}_{out}) \approx f_{NLS}(t, p, z_{in}) = z_{out} \tag{8}$$

is defined, where the solution from the previous time step $\hat{z}_{out}$ is given as an additional input. With the information from the previous ODE integrator step the surrogate should learn to predict a solution that is close to the previous solution and not jump to a different solution branch.

3.4 Active Learning

Data for the NLS can be generated at arbitrary inputs inside an appropriate region as discussed in Section 3.2. However, a call to the root finding algorithm is expensive in general. Therefore, a variation of active learning (AL) (Settles 2009; Wu, Lin, and Huang 2018) can be used for training the surrogate f_S.

The general idea of AL is to let the surrogate decide which samples to label, i.e. which inputs to generate the corresponding outputs for. Between training steps the performance of f_S is tested on new inputs z_{in}. Samples from unfit inputs i.e., inputs for which f_S performs poorly, are added to the set T for the next training step. This is described more precisely in Algorithm 2, where m is the number of active learning steps, n is the number of total samples to generate, and $1 - p$ is the fraction of samples that are generated randomly for the first training step, so $p = 0$ is equivalent to not using AL.

Algorithm 2 Active Learning

1: **procedure** ACTIVELEARN(m, n, p)
2: $T \leftarrow$ initial data set with $|T| = (1 - p)n$
3: **for** $i = 1, \ldots, m$ **do**
4: train f_S on T
5: $T' \leftarrow$ FINDUNFITSAMPLES($\frac{pn}{m}$)
6: $T \leftarrow T \cup T'$
7: **return** f_S
8: **procedure** FINDUNFITSAMPLES(n)
9: $T \leftarrow \emptyset$
10: **for** $i = 1, \ldots, n$ **do**
11: choose $z_{in} \in I_{in}$ with large expected error
12: $z_{out} \leftarrow f_{NLS}(t, p, z_{in})$
13: $T \leftarrow T \cup \{(z_{in}, z_{out})\}$
14: **return** T

If Equation 6 does not apply, the residual norm

$$\tau_{abs}(z_{in}, f_S) := \|f_{res}(t, p, z_{in}, f_S(z_{in}))\|_2 \tag{9}$$

gives a comparatively cheap measure for identifying unfit inputs, without the need for root finding. However, residual equations may still contain expensive computations, so in either case evaluations need to be done economically. Since there is some freedom in generating new data, finding unfit inputs can be seen as a multimodal optimization problem inside the classical ML optimization problem

$$\min_{f_S} \left\{ \max_{z_{in}} \tau_{abs}(z_{in}, f_S) \right\} \tag{10}$$

which can be solved by any cheap optimization heuristic and points generated along the way can be added to the

DOI
10.3384/ecp204275

data set T. This corresponds to line 11 in Algorithm 2. A simple variant of the bees algorithm (Pham et al. 2006) was chosen which combines global and local search.

An analysis of the effectiveness of AL is given in Section 4.3.

3.5 Integrate Surrogate into Simulation

The trained Flux model from Section 3.3 is exported in the Open Neural Network Exchange (ONNX) format (Bai, Lu, Zhang, et al. 2017) using ONNXNaiveNASflux.jl[4]. For each f_{NLS} that is replaced by a surrogate the corresponding ONNX file is copied into the `M.interface.fmu` resources directory. The ONNX Runtime (ORT) (ONNX Runtime developers 2021) is used to interact with the ONNX object. During `fmi2Instantiate` all ORT data is initialized and the ONNX files are loaded. In the C code responsible for evaluating the NLS it is possible to switch between the iterative solver method and the evaluation of the surrogate FNN. The FMU is then compiled and packed into `M.onnx.fmu`.

3.6 Surrogate Error Control

Using Equation 1 it is possible to define an error control algorithm. Computing the residual error from Equation 9 is cheap, but for many examples it is important to use the scaled residual norm instead:

$$\tau_s(J) := \|s(J) \circ f_{res}(t, p, z_{in}, \tilde{z}_{out})\|_2 \qquad (11)$$

Here $\circ$ is element-wise multiplication, s a scaling vector

$$s_i(J) := \frac{1}{\|J_{i,*}\|_\infty}, \quad i = 1, \ldots, n_{out}, \qquad (12)$$

with $J_{i,*}$ being the i-th row of the Jacobian J of f_{NLS} and $\|\cdot\|_\infty$ the maximum norm. To utilize the scaled residual norm for the error control it is necessary to evaluate the Jacobian J at each time step. Especially for numeric Jacobians this can be costly to evaluate, but it is still cheaper than a Newton-Raphson step, where the Jacobian needs to be inverted in addition.

Hoping that the Jacobian does not change too much during simulation, Algorithm 3 reuses J from the initialization and updates it whenever the default iterative method needs to evaluate the Jacobian anyway.

If f_S is not performing well on z_{in} or $z_{in} \notin I_{in}$, Equation 2 can be solved by the iterative solver method with a start value from the surrogate or extrapolated from previous solutions $\hat{z}_{out}$.

3.7 (Re-)Initialization and Events

The Modelica language is able to express models that can have discontinuities in the right-hand side of their ODE system. For $model_1$ from Equation 7 events and reinitialization are no issue, if the event is not changing the system structure of the NLS.

[4]GitHub Repository: DrChainsaw/ONNXNaiveNASflux.jl

Algorithm 3 Error Control

1: **procedure** ERRORCONTROL(τ, J)
2:     **if** $z_{in} \in I_{in}$ **then**
3:         $z_{out} \leftarrow f_S(t, p, z_{in})$
4:         **if** $\tau_s(J) > \tau$ **then**
5:             $z_{out}, J \leftarrow f_{NLS}(t, p, z_{in})$ with start value z_{out}
6:     **else**
7:         $z_{out} \leftarrow extrapolate(\hat{z}_{out})$
            ▷ extrapolate z_{out} from previous time step(s)
8:         $z_{out}, J \leftarrow f_{NLS}(t, p, z_{in})$ with start value z_{out}

In contrast $model_2$ from Equation 8 uses the solution $\hat{z}_{out}$ from the previous time step. If the NLS is solved for the first time or if the previous solution is invalid because an event occurred the previous solution is not available and the original equation f_{NLS} is evaluated first.

The `delay` and `spatialDistribution` operators of the Modelica language specification are not considered but they don't seem to be an issue as long as they are not used inside f_{res}.

4 Experiments

The method described in section 3 is tested on a mechanical mass-spring system with scalable non-linear equation systems. The library is presented in Section 4.2 and the generation of surrogates is described in Section 4.2.1. In Section 4.2.2 the simulation results are compared to the Newton-Raphson method used by OpenModelica. Section 4.3 demonstrates reduced data consumption on a Modelica toy model.

4.1 Test Setup

All examples were run on a test server with an Intel Xeon Gold 6248R CPU @ 3.00GHz, 192 GB DDR4 RAM @ 2933 MT/s, NVIDIA Quadro RTX 6000 GPU with 24 GB SDRAM on Ubuntu 22.04.2 LTS. Julia v1.9.0 with packages FMI.jl v0.12.2, Flux.jl v0.13.16, ONNXNaiveNASflux.jl v0.2.7, OMJulia.jl v0.2.1 together with OpenModelica v1.22.0-dev-156 was used.

4.2 ScalableTranslationStatistics Library

In this section the Modelica library ScalableTranslationStatistics[5] is presented, which will be used as an example for the algebraic loop replacement in Section 4.2.1. The ScalableTranslationStatistics library offers many possibilities to create models of specified numerical complexity. It can be used to create generic examples for specific numeric problem classes and thus avoids the need of sharing confidential models.

Model properties like number of algebraic loops, continuous time states or use of numeric Jacobians can be specified a priori via structural parameters in the model. These are the structural properties the model will have after the translation. Due to different algorithms used during

[5]ltx.de/download/ModelicaLibraries/ScalableTranslationStatistics

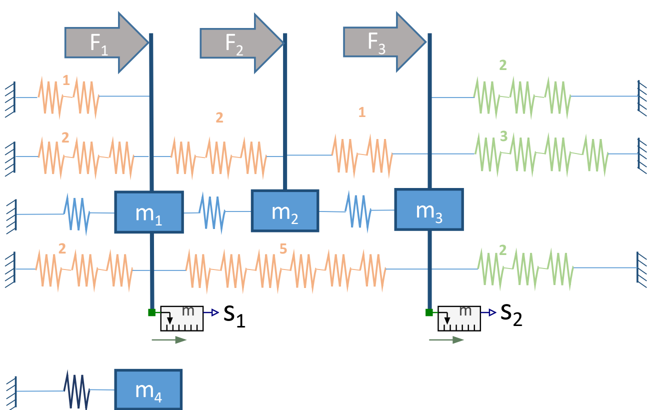


Figure 1. Scalable mass-spring system with parametrization `num_masses=4`, `Lin_equations = {2,3,2}` and `NL_equations = {2,1,5,1,2,2}`.

the translation in different Modelica tools, the final structural properties might slightly differ from the given parameters, but the principal functionality is not tool-dependent.

The principal idea of this library - to scale models - is based on the ScalableTestSuite[6]. But since this library doesn't offer the possibility to scale structural properties like the size of algebraic loops, the ScalableTranslationStatistics library was developed. Figure 1 shows the physical representation for an exemplary parametrization of the model.

A specified number of continuous time states is reached by introducing masses, each having two state variables (position and velocity). In the most simple case the masses are only connected with simple linear springs (blue springs in Figure 1), to avoid in any case singular systems due to free, unconnected masses. To obtain linear or nonlinear equation systems, springs are directly connected with each other. Thus, an algebraic loop of size $M-1$ is created, where M is the number of springs. Depending on the spring characteristics the algebraic loop has a linear or nonlinear behavior.

A simple way to enforce numeric block Jacobian is to introduce an assert-statement in the spring-characteristic to limit the force to a given maximum value, or reading the characteristics from a file. In both cases the characteristic cannot be differentiated analytically but needs to be solved numerically. In the latter case an arbitrary characteristic can be defined.

Furthermore, following additional features are implemented:

- External forces (F_i) can act as inputs to the system, position measurement sensors (S_i) as outputs. By an appropriate definition of the input-forces (e.g. via a TimeTable) a desired number of time- or state-events can be reached.
- Additional parameters, time varying variables and alias variables can be added to the model. They have no influence on the physical behavior, but increase the size of the model.
- The model contains an independent mass-spring system with a different stiffness of the spring (m_4 in Figure 1). Thus, the stiffness can be adjusted to increase the effort for an integrator to solve the model.
- Two-dimensional springs can be added to the model. The directional stiffness of these springs depends on the deflection in both directions. In this way partial derivatives are introduced.

Besides the principal model, the library offers numerous examples of different scaling and for the above-mentioned features.

4.2.1 Generating Surrogates

To test the method presented in section 3 the model `ScalableTranslationStatistics.Examples.ScaledNLEquations.NLEquations_N` is used with $N \in \{5, 10, 20, 40\}$ where N scales the number of iteration variables. The model has eight NLSs with $2N$ unknowns. Because of tearing (Täuber et al. 2014) there are N iteration variables and N inner variables. The first seven systems cannot be differentiated symbolically, therefore numeric Jacobians are used. The fastest system is differentiated symbolically. ODE integrator DASSL (Petzold 1982) is used with tolerance 10^{-6} to simulate in $I_t = [0, 10]$. For small systems ($N \in \{5, 10, 20\}$) the dense Newton-Raphson method is used. For the larger systems ($N \geq 40$) the sparsity of J is 7 %, so the sparse solver KINSOL (Alan C Hindmarsh et al. 2005; Alan C. Hindmarsh et al. 2023) is used instead of the dense one. After profiling Figure 2 shows a significant amount of the simulation time is spent to solve seven of the NLSs, which is an upper bound for the amount of time that could be saved by a faster surrogate.

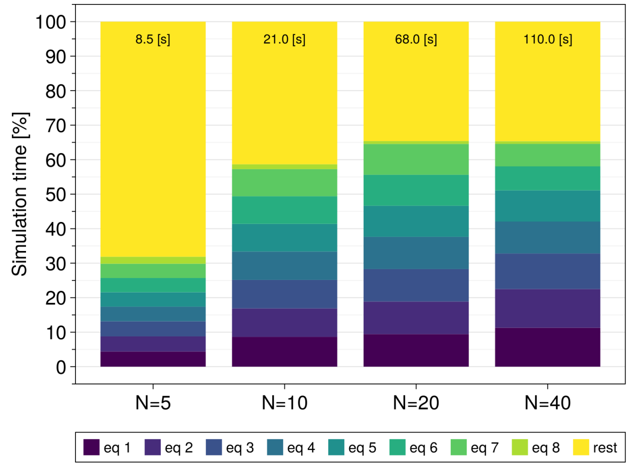


Figure 2. Profiling relative simulation times for `ScalableTranslationStatistics.Examples.ScaledNLEquations.NLEquations_N` with $N \in \{5, 10, 20, 40\}$ and the eight NLS *eq* $1, \dots,$ *eq* 8 as well as all remaining equations *rest*.

[6] github.com/casella/ScalableTestSuite

DOI
10.3384/ecp204275

During data generation Algorithm 1 is used to generate 5,000 data points in 100 batches of 50 calls to `randomWalk` with $\delta = 0.01$ for each N.

80 % of the available data points are used for the training set, the remaining 20 % for the validation set. The model is simulated over $I_t = [0, 10]$ to test the ANN.

For FNN Equation 7 is fitted to the normalized training data $\tilde{I}_{in}$ using Julia package Flux.jl (Michael Innes et al. 2018; Mike Innes 2018). A model with one input, one hidden and one output layer is created:

```
model1 = Flux.Chain(
  Flux.Dense(nIn,      nIn*10,   σ),
  Flux.Dense(nIn*10,   nOut*10,  tanh),
  Flux.Dense(nOut*10,  nOut)
)
```

with $\texttt{nIn} = 1 + n_{in}$ and $\texttt{nOut} = n_{out}$, activation functions sigmoid σ and hyperbolic tangent tanh. The mean square error is used as loss function and Adaptive Moment Estimation (Adam) optimization is used to train the model.

$model_1$ is trained over 1000 epochs or until the loss of the training set is below 10^{-6}.

4.2.2 Simulation Results

With the generated FMU containing surrogates for all eight NLSs the simulation times are measured and compared to the Newton-Raphson method as reference. The models are simulated with an explicit Euler method with fixed step size of 0.001 in time interval $[0, 10]$. In Figure 3 the simulation times and speedup factors are plotted for different values of N. While the evaluation of $model_1$ is slower for small NLS, with growing N the surrogates are significantly faster to evaluate (up to 9 times). With more sophisticated ANN structures even better performance is expected. Unfortunately, the overall savings for the total simulations are not as large as expected. For $N = 40$ the surrogate is only 1.55 times faster than the original high fidelity model.

In Figure 4 the simulation results of iteration variables `scalableModelicaModel.springChain[1].spring[m].s_rel` for $m \in \{1, 3, 4, 5\}$ from NLS equation 808 are displayed. It can be observed, that while the results of the iteration variables of the surrogate compared to the reference solution have a low absolute error

$$|z_{out_i} - \tilde{z}_{out_i}|, \quad i \in \{1, \ldots, 4\} \tag{13}$$

the error $\tau_s(J)$, displayed in Figure 5, of the residual is relatively large. If $\tau_s(J) > 1$ the original Newton-Raphson method was used to solve the NLS.

The relevant output variables are the positions of eight masses `output[m]` for $m \in \{1, \ldots, 8\}$. The results of the surrogate simulation are compared to the reference solution in Figure 6.

4.3 Controlled Data Generation

The training improvements of AL from Section 3.4 are studied using the simple Modelica model `SimpleLoop`

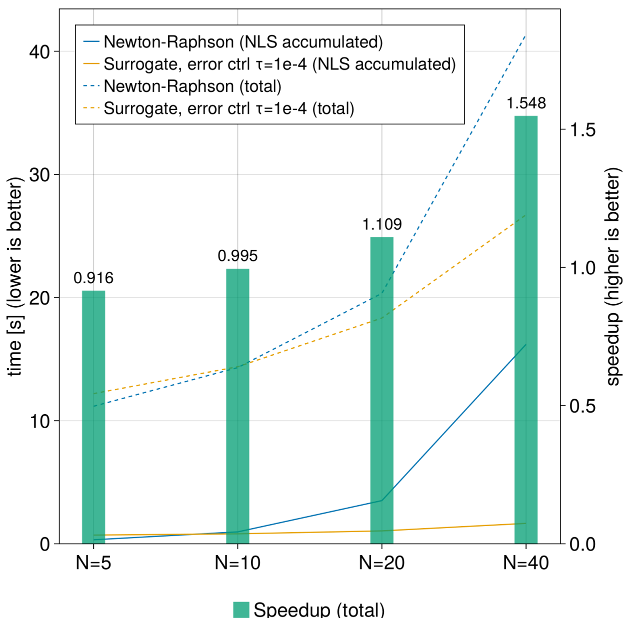


Figure 3. The simulation time of the surrogates are plotted against the reference Newton-Raphson method as well as the respective speedup in total simulation time.

which has a single NLS of size 2. It describes the intersection of a circle with a line, both changing over time:

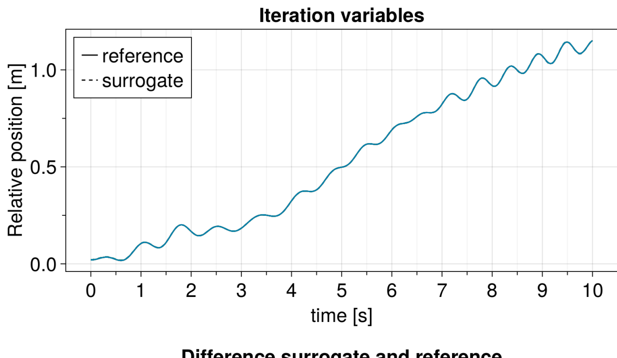
```
model SimpleLoop
  Real r = 1+time;
  Real s = sqrt((2-time)*0.9);
  Real x(start=1.0), y(start=-0.1);
equation
  r^2 = x^2 + y^2;
  r*s = x + y;
  annotation(experiment(StopTime=2));
end SimpleLoop;
```

This model has two distinct solutions since the equations are symmetric in `x` and `y`. The NLS is torn to a single iteration variable and has two inputs, `r` and `s`.

For this model a surrogate with a total of 441 parameters was trained. Figure 7 shows the peaks in τ_{abs} from Equation 9 for different training scenarios. For $p = 0$ surrogates improve slightly with increasing data size n. Using AL on $p = \frac{1}{4}$ of the generated data improves the surrogate significantly, even for small n. Generating $p = \frac{1}{2}$ to $\frac{3}{4}$ with AL seems to be optimal with slight improvements over $p = \frac{1}{4}$. However, surrogates with no pre-training whatsoever perform no better or even worse than with $p = \frac{1}{2}$ pre-training. In particular, for the scenario $n = 800, p = 1$, two out of ten simulations showed bad results and so did one simulation for $n = 1000, p = 0.5$. Still, $p = 0$ produces worse results regardless of the value of n.

5 Results

When increasing the size of the non-linear systems the advantage of surrogates becomes more and more evident.



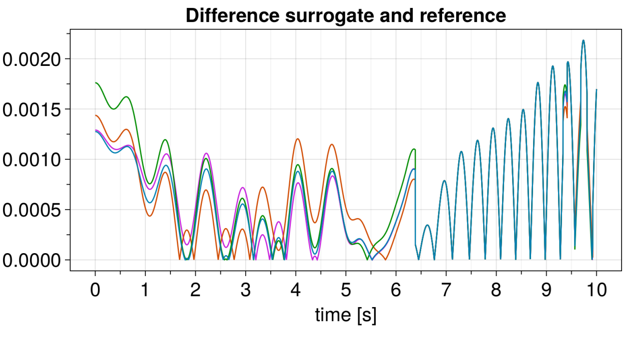


Figure 4. For $N = 5$ the iteration variables of the surrogate and the reference solution are compared in the upper graph. The solid lines are the reference and the dotted lines the surrogate solution. They are so close, that the dotted lines are not visible. The variables are describing a relative position of springs in a chain of springs. The lower graph shows the absolute difference between the reference and surrogate solution.

However, the examined Modelica model is simple. When more complex and application-oriented examples where investigated several problems were encountered, that are not yet solved and discussed in the following subsection.

The AL approach was tested on a small toy example and proved to outperform the method of random data generation both in precision and in data efficiency. However further experiments need to be done to see to what extent AL can impact the surrogate training process for more complex Modelica models.

5.1 Encountered Problems

For larger and highly non-linear NLS it is more complicated to train accurate enough FNN. Simple NLS that have only a few iteration variables can use a lot of previously computed variables. The surrogate tries to approximate a function from the used variables to the solution of the iteration variables. In this case an easy to solve NLS becomes difficult to train for an ANN.

An especially difficult problem are iteration variables that influence the ODE states. There are two different issues with this. When trying to solve the simulation executable with the surrogates the error control cannot man-

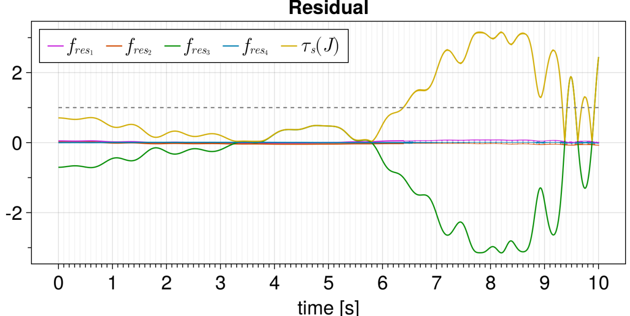


Figure 5. For $N = 5$ equation 808 the residual vector and its norm over the simulation time are displayed. If the value of $\tau_s(J)$ is larger than 1 Algorithm 3 switches to the original non-linear solver method to refine the prediction from the surrogate.

age the imprecise solution of the surrogate. While the approximation would be good enough for the use case the error control of the ODE method will reduce the step size until the lower limit is reached. And when the error control of ODE integrator is deactivated coupled states are a problem.

For analytic stable ODE a well-suited integration method transports this stability to the numeric approximation of the solution. This means numeric errors will vanish over time and the numeric solution converges to the analytic solution. In contrast it seems that small errors from the surrogate will escalate over time and the numeric solution gets worse over time. This needs to be investigated more. Iteration variables that have a high sensitivity towards states can be a significant issue.

6 Summary and Outlook

The presented prototype NonLinearSystemNeuralNetworkFMU.jl aims to lower the bar for Modelica users to utilize hybrid modeling approaches in their models. Even though the scripts are in an early development stage they could pave the way for tool supported integration of ML into Modelica models.

The example from section 4 shows that there is some potential in replacing sufficient large NLS with machine learning surrogates. The focus of this paper is to automate the workflow for data generation and integration of trained surrogates into the simulation executable. So further refinements of the ANN could result in faster and more precise evaluations of the surrogates. The ability to use hybrid ML methods for problems with discrete events and inputs distinguishes the presented method from methods replacing all of the right-hand side of the ODE and ensures correctness of the surrogates.

The workflow can be extended to profile linear equation systems as well as external Modelica functions and automate data generation for these systems or functions. Especially functions computing media properties, e.g. for thermal fluid systems, can be expensive and library developers and users are searching for ways to decrease the

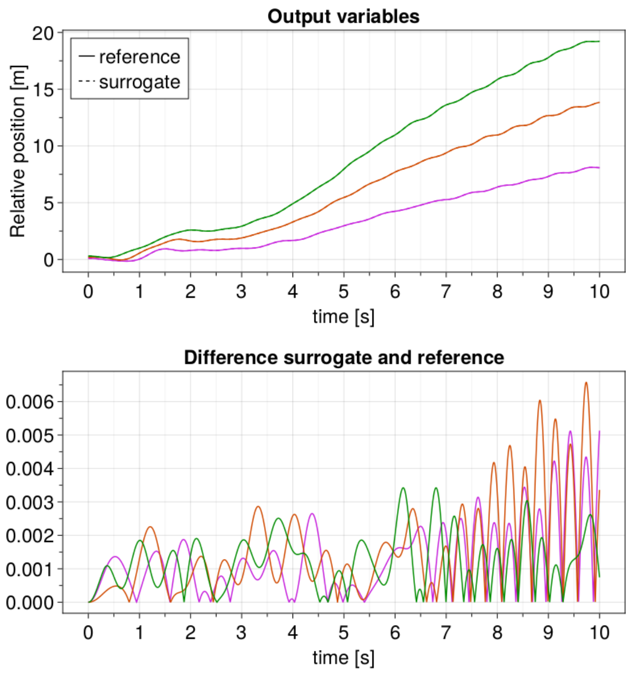


Figure 6. Simulation results in the upper graph and absolute errors in the bottom graph for $N = 5$ equation 808. In the upper graph the solid lines are the reference and the dotted lines the surrogate solution. They are so close, that the dotted lines are not visible.

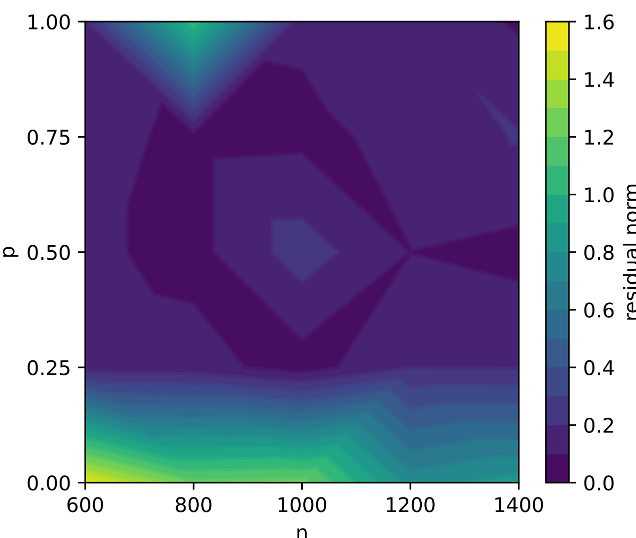


Figure 7. Peaks of τ_{abs} after simulating surrogates trained with $m = 10$ and different values of $n \in \{600, 800, 1000, 1200, 1400\}$ and $p \in \{0, 0.25, 0.5, 0.75, 1\}$. Each data point is the average over 10 surrogates.

time spent evaluating these functions.

Instead of using ANN we plan to use symbolic regression to obtain equations that represent the underlying physics while being less of a black box.

Acknowledgements

This work has been supported by the German Federal Ministry for Economic Affairs and Climate Action in the framework of PHyMoS - Proper Hybrid Models for Smarter Vehicles (grant number 19I20022G and 19I20022E).

References

Anantharaman, Ranjan et al. (2020). *Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks*. DOI: 10.48550/ARXIV.2010.04004. URL: https://arxiv.org/abs/2010.04004.

Bai, Junjie, Fang Lu, Ke Zhang, et al. (2017). *ONNX: Open Neural Network Exchange*. https://github.com/onnx/onnx. Version: v1.12.0.

Fritzson, Peter et al. (2020). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control* 41.4, pp. 241–295. DOI: 10.4173/mic.2020.4.1.

Hindmarsh, Alan C et al. (2005). "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3, pp. 363–396. DOI: 10.1145/1089014.1089020.

Hindmarsh, Alan C. et al. (2023). *User Documentation for KINSOL*. v6.5.1.

Hochreiter, Sepp and Jürgen Schmidhuber (1997-11). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Hübel, Moritz et al. (2022-11). "Hybrid physical-AI based system modeling and simulation approach demonstrated on an automotive fuel cell". In: *Modelica Conferences*. Ed. by Tielong Shen, Rui Gao, and Yutaka Hirano. Linköping University Electronic Press, pp. 157–163. DOI: https://doi.org/10.3384/ecp193157.

Innes, Michael et al. (2018). "Fashionable Modelling with Flux". In: *CoRR* abs/1811.01457. arXiv: 1811.01457. URL: https://arxiv.org/abs/1811.01457.

Innes, Mike (2018). "Flux: Elegant Machine Learning with Julia". In: *Journal of Open Source Software*. DOI: 10.21105/joss.00602.

Lawal, Zaharaddeen Karami et al. (2022). "Physics-Informed Neural Network (PINN) Evolution and Beyond: A Systematic Literature Review and Bibliometric Analysis". In: *Big Data and Cognitive Computing* 6.4. ISSN: 2504-2289. DOI: 10.3390/bdcc6040140. URL: https://www.mdpi.com/2504-2289/6/4/140.

Mattsson, Sven Erik and Hilding Elmqvist (1997). "Modelica-An international effort to design the next generation modeling language". In: *IFAC Proceedings Volumes* 30.4, pp. 151–155.

ONNX Runtime developers (2021). *ONNX Runtime*. https://onnxruntime.ai/. Version: 1.12.1.

Petzold, Linda R (1982). *Description of DASSL: a differential/algebraic system solver*. Tech. rep. Sandia National Labs., Livermore, CA (USA).

Pham, D.T. et al. (2006). "The Bees Algorithm — A Novel Tool for Complex Optimisation Problems". In: *Intelligent Production Machines and Systems*. Ed. by D.T. Pham, E.E. Eldukhri, and A.J. Soroka. Oxford: Elsevier Science Ltd,

pp. 454–459. ISBN: 978-0-08-045157-2. DOI: https://doi.org/10.1016/B978-008045157-2/50081-X. URL: https://www.sciencedirect.com/science/article/pii/B978008045157250081X.

Settles, Burr (2009). *Active Learning Literature Survey*. Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

Sjölund, Martin (2015). *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models*. Vol. 1664. Linköping University Electronic Press.

Täuber, Patrick et al. (2014). "Practical Realization and Adaptation of Cellier's Tearing Method". In: *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '14. Berlin, Germany: Association for Computing Machinery, pp. 11–19. ISBN: 9781450329538. DOI: 10.1145/2666202.2666204. URL: https://doi.org/10.1145/2666202.2666204.

Thummerer, Tobias, Lars Mikelsons, and Josef Kircher (2021-09). "NeuralFMU: Towards Structural Integration of FMUs into NeuralNetworks". In: *Proceedings of 14th Modelica Conference 2021*. Ed. by Adrian Pop Martin Sjölund Lena Buffoni and Lennart Ochel. Linköping University Electronic Press, pp. 297–306. DOI: 10.3384/ecp21181297.

Wu, Dongrui, Chin-Teng Lin, and Jian Huang (2018). *Active Learning for Regression Using Greedy Sampling*. arXiv: 1808.04245 [cs.LG].