Hochschule Bielefeld
University of Applied Sciences

Institute for Data Science Solutions of HSBI

Residual-informed Learning of Solutions to Algebraic Loops

Schriftenreihe des Institute for Data Science Solutions

Nr. 3/ 2025

DOI: 10.60802/sidas.2025.3
ISSN: 2943-3509

Residual-informed Learning of Solutions to Algebraic Loops

Felix Brandt[a], **Andreas Heuermann, Philip Hannebohm**[c] **und Bernhard Bachmann**[d]

[a,c,d]Institute for Data Science Solutions – Bielefeld University of Applied Sciences and Arts, Germany,
{felix.brandt,philip.hannebohm,bernhard.bachmann}@hsbi.de
ORCiD ID: Andreas Heuermann https://orcid.org/0009-0000-1792-1701,
Philip Hannebohm https://orcid.org/0009-0003-8902-9079,
Bernhard Bachmann https://orcid.org/0000-0002-4339-0438

Abstract. This paper presents a residual-informed machine learning approach for replacing algebraic loops in equation-based Modelica models with neural network surrogates. A feedforward neural network is trained using the residual (error) of the algebraic loop directly in its loss function, eliminating the need for a supervised dataset. This training strategy also resolves the issue of ambiguous solutions, allowing the surrogate to converge to a consistent solution rather than averaging multiple valid ones. Applied to the large-scale IEEE 14-Bus system, our method achieves a ∼60% reduction in simulation time compared to conventional simulations, while maintaining the same level of accuracy through error control mechanisms.

1 Introduction

Simulating Modelica models can be computationally intensive in the presence of non-linear equation systems (NLSs) which typically require iterative methods, such as the Newton-Raphson algorithm, to solve. As these systems grow larger, the computational cost of solving them becomes significant.

While artificial neural network (ANN) surrogates offer fast prediction speed, providing and processing enough high quality labeled training data can be an issue and prohibit high accuracy prediction results. Reasons can be the size of the dataset, lack of computational resources and/or ambiguities in the provided dataset. The latter is a big problem in our application.

The residual formulation of an NLS quantifies the deviation between a proposed solution and a true solution. This information can be used in the loss function of an ANN in order to measure goodness of fit, while not requiring a labeled dataset at all. When the residual is used in the loss function, the ANN learns to predict the solution vector from an initial guess vector—i.e., a set of approximate values for the algebraic iteration variables—for the algebraic loop.

Still, training an accurate ANN requires computational resources. Building such surrogate models is thus particularly useful when a Modelica model needs to be simulated many times, such as in optimization or control tasks.

Related Work

A working automated pipeline for creating surrogate models was presented in [3]. In that approach, NLSs are replaced with ANNs trained on labeled datasets consisting of input–output pairs. Additionally, they introduced an error control mechanism that falls back to Newton-Raphson when the ANN prediction error exceeds a predefined threshold. In contrast, this work removes the need for a labeled dataset entirely.

Training a Physics-Informed Neural Network (PINN) [9] involves embedding the residual of a partial differential equation (PDE) into the loss function. Our approach is conceptually similar, but instead of using PDE residuals, we use the residuals of algebraic equations that appear in the right-hand side (RHS) computation of ordinary differential equations (ODEs).

As defined in [6], a Physics-Enhanced Neural ODE (PeNODE) represents a meaningful combination of physical equations and ANNs. Our work can be viewed as a special case of PeNODEs, where the focus lies specifically on replacing algebraic equations with ANNs in a physically consistent manner.

2 Problem Statement

In a Modelica simulation context an algebraic loop is given in residual form

$$\begin{aligned} f &: \mathbb{R}^{n_{in}} \times \mathbb{R}^{n_{out}} \to \mathbb{R}^{n_{out}}, \\ f(\mathbf{x}, \mathbf{y}) &= \mathbf{0}, \end{aligned} \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^{n_{in}}$ contains the simulation time, constant system parameters and other used variables computed in preceding model equations, and $\mathbf{y} \in \mathbb{R}^{n_{out}}$ are unknowns for which the algebraic loop is solved. The algebraic loop is solved repeatedly over the course of a simulation with different $\mathbf{x}$. To solve Equation 1, an iterative algorithm like Newton's Method is used, which results in

$$\mathbf{y}_{k+1} := \mathbf{y}_k - J_f(\mathbf{x}, \mathbf{y}_k)^{-1} f(\mathbf{x}, \mathbf{y}_k), \tag{2}$$

where k is the current iteration index and $J_f(\mathbf{x}, \mathbf{y})^{-1}$ is the inverse of the Jacobian of f w.r.t. $\mathbf{y}$. While Newton's method is effective, it requires repeated computation of the Jacobian and residuals, which becomes increasingly costly for large systems or systems with complex algebraic loops.

The primary challenge lies in the computational overhead associated with solving non-linear algebraic loops iteratively. This can significantly slow down simulations, especially when the system must be simulated repeatedly for tasks such as optimization or control, thereby

motivating the need for more efficient surrogate models.

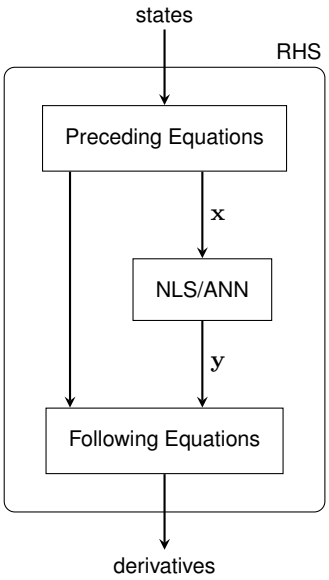

Figure 1: The big box represents the computation of the RHS which takes states as inputs and computes the corresponding state derivatives. It contains computation of preceding and following equations and specifically one NLS, that will be replaced by an ANN. The NLS receives $\mathbf{x}$ from preceding computations inside the RHS and returns $\mathbf{y}$.

Figure 1 illustrates how the ANN replaces the NLS inside the Modelica model.

2.1 Ambiguity Problem

Non-linear algebraic loops can exhibit multiple (ambiguous) solutions, meaning that for a given $\mathbf{x}$, there may be multiple vectors $\mathbf{y}$ satisfying $f(\mathbf{x}, \mathbf{y}) = 0$. In such cases, f does not define a functional relationship from $\mathbf{x}$ to $\mathbf{y}$, as it fails to produce a unique output for each input.

This ambiguity poses a challenge when training neural networks using a standard mean squared error (MSE) loss, since neural networks are designed to approximate continuous functions that assume a unique mapping from inputs to outputs.

The loss formulation proposed in this work addresses this issue by allowing the network to converge to one of the valid solutions, rather than averaging over multiple possibilities. In contrast, training with an MSE loss in the presence of multiple solutions leads to averaging effects, resulting in poor predictive accuracy. A more detailed discussion and a formal argument can be found in Appendix C.

3 Method

The approach proposed in this paper uses an ANN to predict $\mathbf{y}$ from the input $\mathbf{x}$, aiming to replace the use of Newton's method for solving algebraic loops. The loss function is defined directly in terms of the residual f as

$$L(\hat{\mathbf{y}}) = \frac{1}{2} \left\| f(\mathbf{x}, \hat{\mathbf{y}}) \right\|_2^2 . \tag{3}$$

Here, $\hat{\mathbf{y}}$ denotes the predicted solution vector produced by the ANN, while the true solution vector $\mathbf{y}$ is unknown and not required for training.

This particular form of L is chosen because it leads to a simpler expression for the gradient, facilitating more efficient training. By minimizing L, the ANN learns to predict $\mathbf{y}$ such that the residual $f(\mathbf{x}, \hat{\mathbf{y}})$ approaches zero.

The specific solution to which the network converges when using Equation 3 is influenced by factors such as the random initialization of network weights, the learning rate, and other training parameters—analogous to how Newton's method converges to a particular solution depending on the initial guess vector.

For gradient based optimization such as ANN training, one needs to compute the gradient of L w.r.t. $\hat{\mathbf{y}}$. We use the Julia programming language, where automatic differentiation systems like Zygote.jl [4] can handle pure Julia code efficiently. However, these systems fail when computations involve calls to external functions. In such cases, the gradient must be derived and implemented manually. In this study, f is given as C function called from Julia, necessitating the manual implementation of the gradient of L. The gradient of L is

$$\nabla L(\hat{\mathbf{y}}) = J_f^T(\mathbf{x}, \hat{\mathbf{y}}) f(\mathbf{x}, \hat{\mathbf{y}}), \tag{4}$$

with $J_f^T \in \mathbb{R}^{n_{out} \times n_{out}}$ being the transpose of the Jacobian of f w.r.t. $\hat{\mathbf{y}}$. J_f is also obtained by a C function call. $\nabla L(\hat{\mathbf{y}})$ is of shape $\mathbb{R}^{n_{out}}$. A detailed derivation of Equation 4 is provided in Appendix B. From this point on, we refer to an ANN trained using the loss function in Equation 3 as a *residual-trained model* to emphasize that it learns via residual minimization rather than supervised labels.

An output $\hat{\mathbf{y}}$ from the ANN is considered an accurate prediction if the residual $f(\mathbf{x}, \hat{\mathbf{y}})$ is sufficiently small, which corresponds to the loss being close to zero. There are two common ways to quantify whether a prediction is "close enough" to a solution.

The first is to use an absolute threshold on the loss:

$$L(\hat{\mathbf{y}}) \leq atol,$$

where $atol$ is a user-defined absolute tolerance.

The second approach compares successive predictions over the course of training. Let $\hat{\mathbf{y}}_1$ and $\hat{\mathbf{y}}_2$ be predictions for a fixed test input after two successive training epochs (or separated by several epochs). Convergence is then defined elementwise by:

$$|\hat{\mathbf{y}}_2 - \hat{\mathbf{y}}_1| \leq atol + rtol \cdot |\hat{\mathbf{y}}_1|,$$

where $rtol$ is a user-defined relative tolerance that controls convergence independent of the scale of $\hat{\mathbf{y}}$. This condition must be satisfied for all components to declare convergence. Both criteria are implemented and used during training to monitor prediction accuracy.

In addition to residual-based criteria, we also explored using the number of Newton iterations required to reach convergence as an alternative metric. Since our ultimate goal is to reduce the need for such iterations, this provides a meaningful proxy for prediction quality. This criterion could also be used to guide early stopping, e.g., by halting training once the predicted output reduces the number of required Newton steps below a target threshold.

Empirical results supporting this idea are presented in section 5.

One limitation of the current approach is that residuals are not scaled across dimensions. This can lead to imbalance during training when some residual components dominate the loss due to their larger magnitude. Addressing this issue remains part of future work.

For computational efficiency, $\mathbf{x}$ and $\hat{\mathbf{y}}$ are typically batched into matrices with dimensions $\mathbb{R}^{n_{in} \times n}$ and $\mathbb{R}^{n_{out} \times n}$, respectively.[1] Here n

[1] Following the Julia convention with the batch size in the last dimension.

denotes the batch size, or the number of samples. In this case the loss function becomes

$$L(\hat{\mathbf{y}}) = \frac{1}{2n} \sum_{i=1}^{n} \|f(\mathbf{x}[i], \hat{\mathbf{y}}[i])\|_2^2 . \tag{5}$$

Equation 5 is simply the average of Equation 3 over the batch. In the batched case, the gradient is computed by iterating over the batch, evaluating Equation 4 for each element vector, and storing these intermediate results in a final gradient matrix of shape $\mathbb{R}^{n_{out} \times n}$. Example implementations are provided in Appendix A. An implementation can be found on GitHub[2].

3.1 Semi-Supervised Loss

Our method can be extended by adding a supervised MSE term to Equation 3 like so:

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{\lambda}{2} \|f(\mathbf{x}, \hat{\mathbf{y}})\|_2^2 + \frac{1-\lambda}{2} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 \tag{6}$$

where $\mathbf{y}$ is the target/label vector for $\mathbf{x}$. The hyperparameter $0 \leq \lambda \leq 1$ controls the relative influence of the two loss terms. Flux/Zygote handles the automatic differentiation of this modified loss function natively, once the backpropagation rule for the gradient of the first term is provided, which is given in Equation 4.

This approach can be used to guide the learning process towards a specific solution of f. $\mathbf{y}$ is generated before training with Newton's Method. A well chosen set of target vectors $\mathbf{y}$ can be obtained by clustering the label vectors of a randomly generated dataset. A standard clustering algorithm like k-Means can be used for the task.

It is also possible to start training using a small unambiguous dataset (obtained through clustering) and use only the MSE ($\lambda = 0$) to guide the model towards a preferred direction and then switch the loss function to the residual ($\lambda \to 1$) to allow training on a high amount of input data but at the same time guide the training process, which is not possible when only training with Equation 3. When training like that, we were able to guide the model to a preferred solution in the first phase and in the second phase the model stayed on that solution and reduced the loss further, as seen in Figure 2.

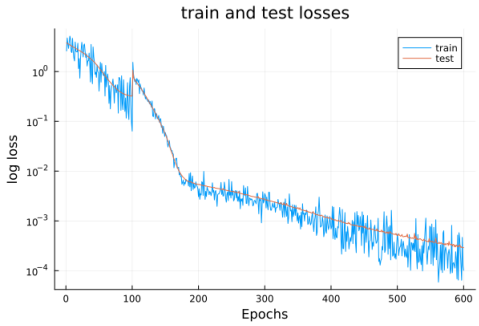

Figure 2: Training loss from a semi-supervised run illustrating the two-phase strategy: MSE loss is used for the first 100 epochs, followed by a residual-based loss. The example is based on the SimpleLoop model from subsection 5.1.

[2] GitHub: AMIT-HSBI/UnsupervisedTrainingExample

4 Connected Solutions

In some cases, the solution set of the NLS is path-connected, meaning that solution branches are not clearly separated but instead form a continuous manifold. This can lead to the simulation transitioning between solutions over time. However, a single ANN can typically represent only one solution branch. To address this, we implemented a proof of concept in which multiple ANNs are trained on different regions of the output space, identified using a clustering algorithm such as k-Means. During simulation, the appropriate ANN is selected based on the input to ensure consistent predictions.

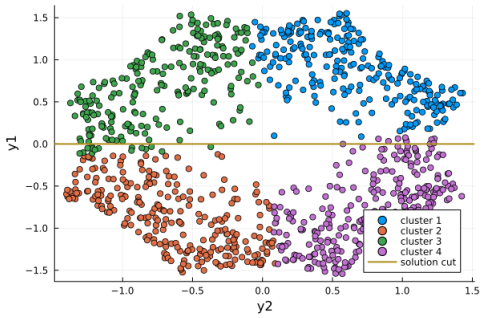

Figure 3: Continuous output space of a model which computes the square root of a complex number, partitioned by k-Means. Now a model is trained on each partition and during the simulation one could then only evaluate the model for the part of output space the simulation is currently in. The model contains two solutions, and the border between them is indicated by the horizontal "solution cut" line.

The appropriate model is selected as the one trained on the region whose centroid is closest to the previous output. An example implementation of this algorithm, along with resulting figures, is provided in Appendix D. Using this approach, we demonstrate that employing multiple models significantly improves the system's capability to handle ambiguous problems with connected solution spaces.

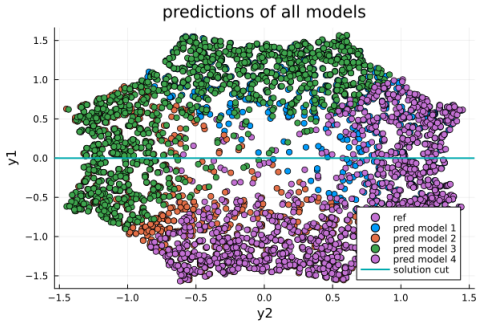

Figure 4: Predictions of all models, each trained on one part of the partition using the second technique from subsection 3.1. One can see that the whole space is covered (with some overlap on the boundaries) and can in principle be predicted.

5 Experiments and Results

Before training, input data of shape $\mathbb{R}^{n_{in} \times N}$ (with $N \approx 10{,}000$ samples) is generated by sampling the input space. To ensure the samples cover the relevant range, a profiling step first simulates the model once, identifying minimum and maximum bounds for each input variable. This guarantees that the training data spans the full input domain encountered during typical simulations. For more details on this

process, see [3], though as discussed later, this method may be improved.

Sampling is performed using the `QuasiMonteCarlo.jl` package [7], which offers advanced sampling methods designed to provide better coverage of high-dimensional spaces than purely random sampling. We used Sobol sampling, a low-discrepancy sequence that ensures more uniform coverage, reducing the number of samples needed for effective training compared to random sampling. Alternative methods like Latin Hypercube sampling are also available and may be suitable depending on the problem.

To assess the model's generalization performance, validation data is generated by simulating the model once before training, providing unseen examples for evaluation during the training process.

The ANN architecture used is a feedforward network with two hidden layers of 160 neurons each, employing ReLU activation functions. This configuration balances expressiveness and computational efficiency: enough parameters to capture complex input-output relationships while maintaining fast inference times. Training is performed using the Adam optimizer over up to 2000 epochs, starting with a learning rate of 8×10^{-4}.

To improve convergence and avoid stagnation, learning rate decay is applied after half the training time. Specifically, the learning rate is multiplied by $\frac{1}{5}$ every 20% of the remaining epochs. For example, in a 2000-epoch training, the decay steps occur at epochs 1000, 1400, and 1800. This gradual reduction helps the model refine its weights during later training stages. While these choices proved effective, there remains significant room for tuning and optimization depending on the specific application.

All training is implemented using the `Flux.jl` package [5] and was run on a CPU (Intel i7-12700T, 1.40 GHz).

The proposed approach is evaluated on two Modelica models that include non-linear algebraic loops. For each model, we assess multiple aspects: the evolution of the training loss, the prediction accuracy during simulation, and the overall simulation time. These results are compared against two baselines: a conventional simulation that solves the algebraic loop using Newton's method, and a supervised surrogate model trained on labeled input-output data. Furthermore, we examine how the approach handles ambiguous solutions, and compare the time required for data generation. In contrast to the supervised method—which requires solving the algebraic loop for each sampled input to generate labels—our method only samples input points, significantly reducing preprocessing cost.

5.1 SimpleLoop

The *SimpleLoop* model describes a growing circle and a moving line, and involves solving a non-linear system to compute the intersection points of the two as they evolve over time. The problem is governed by the following equations:

$$\begin{aligned} x^2 + y^2 &= r^2 \\ x + y &= rs \end{aligned} \tag{7}$$

Here, the first equation defines a circle with radius r, and the second defines a line whose position depends on the parameter s.

The neural network receives r and s as inputs and is trained to predict the y-coordinate of one of the intersection points. The model is simulated over the interval $t \in [0, 2]$, with r and s varying over time as follows: $r = 1 + t$ and $s = \sqrt{0.9(2 - t)}$.

For this example, the residual function f, introduced in Equation 1,

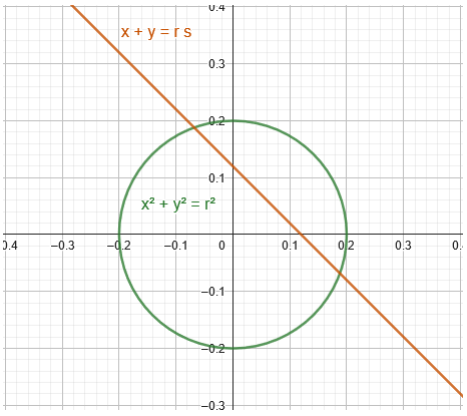

Figure 5: Solution space of Equation 7. As r varies, the circle grows or shrinks and the line moves along. As s varies, the line moves along the main diagonal.

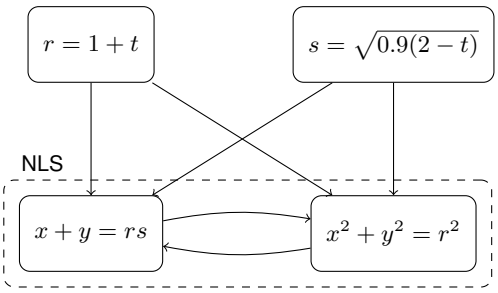

Figure 6: Dependency graph of the SimpleLoop model. It shows the circular dependency of the algebraic loop. The bottom nodes need to be solved simultaneously for x and y. Both r and s are known from the preceding equations.

evaluates to

$$f(\mathbf{x}, \mathbf{y}) = y^2 + (rs - y)^2 - r^2, \tag{8}$$

where $\mathbf{x} = (t, r, s)$, $\mathbf{y} = (y)$, and the substitution $x = rs - y$ has been applied to eliminate x.

This residual formulation results from applying a tearing algorithm, as implemented in OpenModelica [3]. Tearing systematically substitutes equations to reduce the number of unknowns and equations, improving both robustness and efficiency of the numerical solution [2].

This example serves as a minimal test case for evaluating the ability of the residual-based neural network to learn and solve non-linear algebraic systems embedded in dynamic configurations.

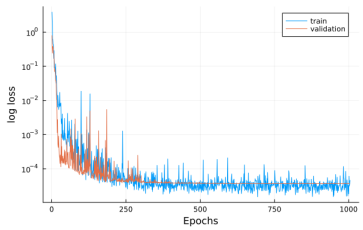

Figure 7: Training and validation loss curve for SimpleLoop.

Figure 7 shows that the model converges early and achieves a training loss of around 1×10^{-4} after 1000 Epochs. There are no

[3] www.openmodelica.org

signs of overfitting on the training data.

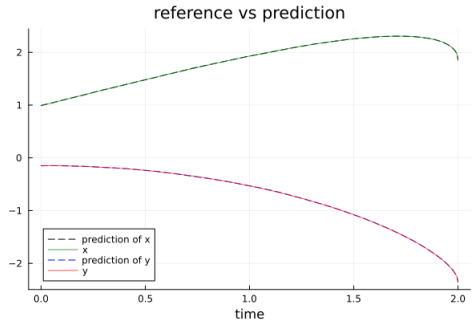

Figure 8: Output variable plot for SimpleLoop. The model outputs a prediction for y and one can compute the corresponding prediction for x.

In Figure 8, the predicted and reference trajectories are visually indistinguishable.

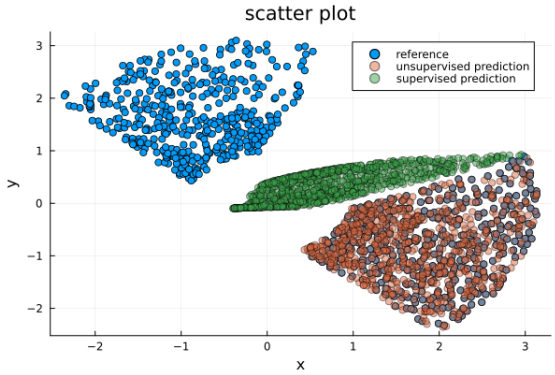

Figure 9: Scatter Residual/Supervised comparison plot for SimpleLoop.

As shown in Figure 9, the supervised model (green) averages between the two true solutions (blue), making it unsuitable for ambiguous problems. In contrast, the residual-based model consistently converges to one of the valid solutions. A potential improvement to the supervised approach could involve clustering the dataset prior to training and training on one of the solution branches individually.

Table 1: Simulation time (in seconds) for the SimpleLoop model versus number of training epochs. *Residual* refers to our method using a surrogate trained with the residual-based loss (Equation 3). *Supervised* uses a surrogate trained on labeled data with an MSE loss. *Classical* denotes the standard simulation using Newton's method. All surrogates were trained on 10,000 data samples.

	Epochs		
Method	10	100	1000
Residual	1.001 s	1.001 s	1.001 s
Supervised	1.001 s	1.001 s	1.001 s
Classical		1.001 s	

Table 1 shows no significant difference in simulation time between the three approaches for this example. To produce the results, three sets of five simulation runs were performed for each number of training epochs and method. The simulation times were measured using Julia's `time()` function and averaged over the five runs.

During surrogate-based simulation, the relative error of the predicted solution is monitored. If the error exceeds a user-defined tolerance, Newton's Method is invoked using the predicted value as the initial guess. This mechanism ensures that surrogate use does not compromise accuracy.

In this particular case, all approaches achieved similar simulation times, indicating no performance gain from the surrogate due to the simplicity of the model.

Table 2: Average data generation time in milliseconds over 5 runs for different numbers of samples N. *Supervised* refers to generating a labeled dataset (input-output pairs), while *Residual* denotes our method, which only samples input points.

	N			
Method	100	1000	10000	100000
Supervised	8340 ms	8410 ms	8580 ms	9010 ms
Residual	**0.0176 ms**	**0.0494 ms**	**0.439 ms**	**7.78 ms**

The residual data generation process involves only sampling input points within the previously determined bounds, which is computationally inexpensive. In contrast, the supervised approach introduces additional overhead, as it requires solving the non-linear system using Newton's method for each sampled input point to generate corresponding output labels. This can be observed in Table 2. While the supervised method does not scale significantly with the number of samples N, its per-sample cost remains higher due to this additional computation.

The proposed residual-based approach accurately captured the behavior of the algebraic loop in the *SimpleLoop* model and yielded satisfactory results in terms of training loss and agreement with reference values. However, as this is a relatively simple, illustrative example, no measurable improvement in simulation time over the classical approach was observed.

5.2 IEEE14

The *IEEE14* bus system is an approximate model of the American electric power grid as it existed in February 1962 [8], and it is included in the OpenIPSL Modelica library [1]. The model contains multiple NLSs that can be targeted for surrogate replacement. In this study, we investigate the substitution of two NLSs of differing sizes to evaluate the performance of our approach on more complex and realistic systems.

5.2.1 Large system

This NLS features a 16-dimensional input vector, referred to as the "used variables" $\mathbf{x}$, and produces a 110-dimensional output vector of "iteration variables" $\mathbf{y}$. To substitute this system, we design a neural network with matching input and output dimensions.

Figure 10 illustrates that the model steadily converges toward a minimum. However, the plot also reveals signs of overfitting and occasional undesired loss spikes.

The plot over time in Figure 11 clearly shows two distinct events corresponding to a ground fault starting at 1 second and lasting for 200 ms. Before the fault, the variable path changes only slightly, appearing almost constant.

The ground fault manifests in the model as the residual depends on an if-statement conditioned on the current simulation time, which introduces another form of ambiguity by switching between branches

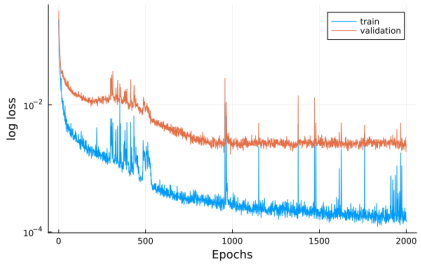

Figure 10: Training and validation loss curve for IEEE14.

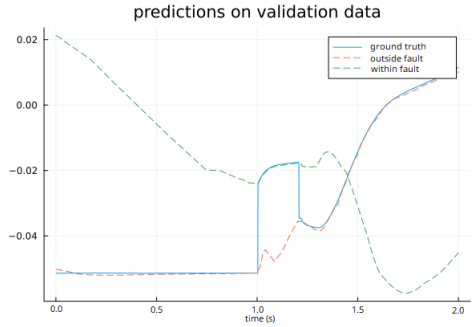

Figure 11: Time series *ground truth* of one reference iteration variable compared to predictions from two neural networks: Model *outside fault* trained only outside the ground fault and model *within fault* trained during the simulated ground fault.

of the piecewise-defined equation. To handle this, we train two separate neural networks: one with the ground fault switched off, and another during the fault. Consequently each network models a different branch of the if-statement.

The results, also shown in Figure 11, demonstrate that one model accurately predicts behavior during the ground fault, while the other performs well before and after it. The final prediction is constructed by combining the outputs of both networks appropriately.

During actual simulation, the appropriate model would be selected dynamically based on the same condition used in the original residual formulation.

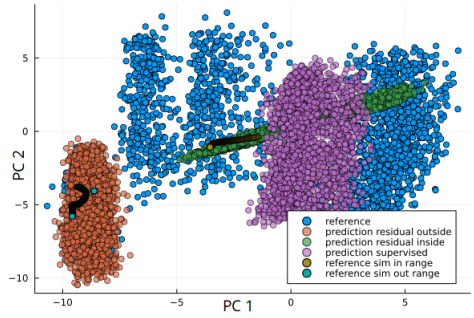

Figure 12: PCA plot of the large IEEE14 system's output space, comparing reference data with residual and supervised predictions. The residual prediction is separated into outputs from the models trained during and outside the ground fault. The output space dimensionality was reduced to two principal components using PCA. Also shown are the data points traversed by a reference simulation.

Figure 12 reveals that the simulation data clusters with principal component analysis (PCA) into two distinct groups, supporting the use of separate networks to handle the ground fault scenario. The supervised model's predictions, however, do not converge to a single solution; instead, they spread between clusters, indicating ambiguity. In contrast, each unsupervised (residual) model converges clearly to one solution cluster without averaging across clusters. This suggests that our method effectively resolves ambiguity both in low- and high-dimensional spaces.

Additionally, it is evident that many reference data points contribute little to training, highlighting the potential benefit of a more targeted data generation strategy focused on regions near actual simulation trajectories.

Table 3 shows the progression of the number of Newton iterations required for convergence throughout the training process. The first data point corresponds to the completely untrained network, and subsequent measurements are taken every 50 epochs during training.

Table 3: Number of Newton iterations to converge when initialized with the model prediction. The first column shows iterations starting from a random initial guess without model prediction. Subsequent values represent the mean iterations over a batch of inputs, typically between 3 and 4.

	Num Epochs			
Model	0	100	500	1000
Large system	186.7	4.0	3.7	**3.3**

In Table 3, a clear improvement is observed over the course of training, with the number of Newton iterations converging to fewer than five. Notably, after an initial drop, the required iterations stabilize, suggesting that stopping training early would not significantly affect the eventual speedup in simulation time. Consequently, whether training lasts for 100 or 1000 epochs may have little impact if the primary goal is to improve simulation efficiency.

Table 4: Simulation time in seconds versus number of training epochs for the IEEE14 model. Classical refers to conventional simulation; Surrogate denotes our method. Training used 5000 data samples.

	Epochs		
Method	10	100	1000
Residual	**10.0 s**	**10.0 s**	**10.2 s**
Supervised	14.0 s	**10.0 s**	**10.2 s**
Classical		28.0 s	

Table 4 demonstrates that simulation using the surrogate model is approximately 60% faster than the conventional approach.

Figure 13 compares the training and validation loss curves of the supervised and residual methods. The residual method converges to a lower loss than the supervised method. This is expected, as shown in e.g. Figure 9.

Table 5: Data generation time (in seconds) for varying sample sizes N. Residual denotes our method, while Supervised refers to generating labeled datasets.

	N		
Method	100	1000	10000
Supervised	44.347	58.047	449.467
Residual	**0.0005**	**0.0003**	**0.0003**

Table 5 shows that supervised data generation requires significantly more time, as expected. Due to the high dimensionality of the

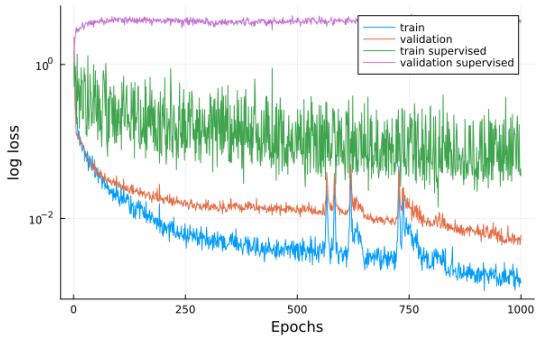

Figure 13: Comparison of training and validation loss progressions for supervised and residual methods. The residual approach clearly outperforms the supervised method, as expected.

system, issues such as NaN and infinite values during Newton iterations occur more frequently, leading to prolonged generation times.

5.2.2 Smaller system

Here, we replace a smaller algebraic loop within the IEEE14 model, characterized by four input variables and a single output.

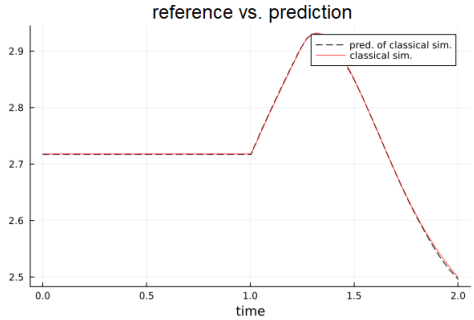

Figure 14: Prediction versus reference plot of the single output dimension over simulation time.

Figure 14 shows high prediction accuracy, with the surrogate model closely matching the reference output throughout the simulation. This indicates that the network effectively captures the system's nonlinear behavior for this smaller algebraic loop, demonstrating the capability of the proposed residual training approach in delivering precise predictions even in low-dimensional cases.

6 Summary and Outlook

This publication explored the application of a residual loss approach to train neural networks for learning algebraic loops. The method proved successful on both small-scale toy problems, such as *SimpleLoop*, and more complex systems like the *IEEE 14-bus* network, achieving accurate predictions and improved simulation performance.

A key strength of this approach lies in its ability to eliminate the need for labeled datasets, thereby significantly accelerating the data preparation phase. Furthermore, it effectively addresses the ambiguity problem often encountered when using supervised loss functions like MSE, particularly in systems that admit multiple valid solutions.

Several directions for further improvement emerged from this work. One promising enhancement involves adaptive sampling, where input points are actively selected in regions of high error to increase accuracy and better capture complex behaviors in the input space. Another avenue is domain decomposition, which would partition the input domain into subregions and train specialized models for each, potentially reducing model complexity and enhancing generalization.

Additionally, guiding the sampling process with reference trajectories—those closer to actual simulation behavior—may yield more data-efficient training and faster convergence. Addressing overfitting remains an important concern, especially in larger or more ambiguous systems, where generalization is critical to ensure robustness. Lastly, the current training process does not include residual scaling; incorporating appropriate scaling of residual terms, particularly when dealing with equations of differing magnitudes or units, could improve training stability, convergence speed, and overall accuracy.

Future work will focus on these directions to further refine the proposed method and extend its applicability to real-world simulation tasks involving algebraic loops. These advancements hold the potential to improve both simulation fidelity and computational efficiency in complex hybrid modeling environments.

References

[1] Maxime Baudette, Marcelo Castro, Tin Rabuzin, Jan Lavenius, Tetiana Bogodorova, and Luigi Vanfretti, 'Openipsl: Open-instance power system library — update 1.5 to "itesla power systems library (ipsl): A modelica library for phasor time-domain simulations"', *SoftwareX*, **7**, 34–36, (2018).

[2] Andreas Heuermann and Bernhard Bachmann. Efficient minimal tearing of hybrid algebraic loops for largescale system simulation. OpenModelica Annual Workshop https://openmodelica.org/images/M_images/OpenModelicaWorkshop_2020/MinimalTearing.pdf. Accessed: 2025-07-24.

[3] Andreas Heuermann, Philip Hannebohm, Matthias Schäfer, and Bernhard Bachmann, 'Accelerating the simulation of equation-based models by replacing non-linear algebraic loops with error-controlled machine learning surrogates', in *Proceedings of the 15th International Modelica Conference 2023*, pp. 275–284, (12 2023).

[4] Michael Innes, 'Don't unroll adjoint: Differentiating ssa-form programs', *CoRR*, **abs/1810.07951**, (2018).

[5] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah, 'Fashionable modelling with flux', *CoRR*, **abs/1811.01457**, (2018).

[6] Tobias Kamp, Johannes Ultsch, and Jonathan Brembeck, 'Closing the sim-to-real gap with physics-enhanced neural odes', in *20th International Conference on Informatics in Control, Automation and Robotics, ICINCO 2023*, eds., Guiseppina Gini, Henk Nijmeijer, and Dimitar Filev, volume 2 of *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics*, pp. 77–84. SCITEPRESS, (November 2023).

[7] Christopher Pal, Christopher Rackauckas, et al. Quasimontecarlo.jl: Quasi monte carlo sampling for julia. https://github.com/SciML/QuasiMonteCarlo.jl, 2020. Version 0.3.3.

[8] Power Systems Test Case Archive. IEEE 14-bus system. http://labs.ece.uw.edu/pstca/pf14/pg_tca14bus.htm, 1993. Accessed: 2025-07-24.

[9] M. Raissi, P. Perdikaris, and G.E. Karniadakis, 'Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations', *Journal of Computational Physics*, **378**, 686–707, (2019).

A Example Implementation

Example implementation using Julia v1.10.4 and ChainRulesCore.jl[4] v1.24.0

```julia
using ChainRulesCore

function fres(zout)
  # implements the residual of a non-linear algebraic loop as a function of zout
end

function J_fres(zout)
  # implements the Jacobian of the residual of a non-linear algebraic loop wrt. zout
end

# loss function
function loss(zout_hat)
  # size(zout_hat) = (nout, bs)
  bs = size(zout_hat, 2)
  residuals = Vector{Vector{Float64}}(undef, bs)
  # evaluate residual for each batch element
  for i in 1:batch_size
    residuals[j] = fres(zout_hat[:, i])
  end
  # compute final loss value based on residual vectors
  return 1 / (2 * bs) * sum(norm.(residuals) .^ 2)
end

# gradient of loss function using ChainRulesCore.jl
function ChainRulesCore.rrule(::typeof(loss), zout_hat)
  nout, bs = size(zout_hat)
  l = loss(zout_hat)

  function loss_pullback(l_bar)
    factor = l_bar ./ bs
    zout_hat_bar = Array{Float64}(undef, nout, bs)
    # compute gradient of fres for each batch element
    for i in 1:bs
      zout_hat_bar[:, i] = transpose(J_fres(zout_hat[:, i])) * fres(zout_hat[:, i])
    end
    zout_hat_bar .*= factor

    return (NoTangent(), zout_hat_bar)
  end

  return l, loss_pullback
end
```

B Gradient of Residual Loss Function

Derivation of Equation 4:

$$L(\hat{\mathbf{y}}) = \frac{1}{2}\left\|f(\mathbf{x},\hat{\mathbf{y}})\right\|_2^2 = \frac{1}{2}\left(\sqrt{\sum_{i=1}^{n} f_i(\mathbf{x},\hat{\mathbf{y}})^2}\right)^2 = \frac{1}{2}\sum_{i=1}^{n} f_i(\mathbf{x},\hat{\mathbf{y}})^2$$

Taking the derivative w.r.t. $\hat{\mathbf{y}}$ gives

$$\begin{aligned}
\nabla L(\hat{\mathbf{y}}) &= \nabla\left(\frac{1}{2}\sum_{i=1}^{n} f_i(\mathbf{x},\hat{\mathbf{y}})^2\right) = \frac{1}{2}\sum_{i=1}^{n} 2 f_i(\mathbf{x},\hat{\mathbf{y}})\nabla f_i(\mathbf{x},\hat{\mathbf{y}}) = \sum_{i=1}^{n} f_i(\mathbf{x},\hat{\mathbf{y}})\nabla f_i(\mathbf{x},\hat{\mathbf{y}}) \\
&= f_1(\mathbf{x},\hat{\mathbf{y}})\begin{pmatrix}\frac{\partial f_1}{\partial \hat{y}_1}\\ \vdots \\ \frac{\partial f_1}{\partial \hat{y}_n}\end{pmatrix} + \cdots + f_n(\mathbf{x},\hat{\mathbf{y}})\begin{pmatrix}\frac{\partial f_n}{\partial \hat{y}_1}\\ \vdots \\ \frac{\partial f_n}{\partial \hat{y}_n}\end{pmatrix} = \begin{pmatrix}\frac{\partial f_1}{\partial \hat{y}_1} & \cdots & \frac{\partial f_n}{\partial \hat{y}_1}\\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial \hat{y}_n} & \cdots & \frac{\partial f_n}{\partial \hat{y}_n}\end{pmatrix}\begin{pmatrix} f_1(\mathbf{x},\hat{\mathbf{y}})\\ \vdots \\ f_n(\mathbf{x},\hat{\mathbf{y}})\end{pmatrix} \\
&= J_f^T(\mathbf{x},\hat{\mathbf{y}}) f(\mathbf{x},\hat{\mathbf{y}})
\end{aligned}$$

[4] https://github.com/JuliaDiff/ChainRulesCore.jl

C Ambiguity discussion

Consider an ambiguous dataset $D = \{(x, y_i)\}_{i=1}^n$, where n is the number of data points in D, and there exist multiple distinct target values y_i corresponding to the same input x, i.e., there exist at least two indices $i \neq j$ such that $y_i \neq y_j$.

We argue that a multilayer perceptron (MLP) f_θ trained on D using a mean squared error (MSE) loss cannot achieve arbitrary precision. Instead, its prediction for x will converge to the average of all y_i, i.e.,

$$\hat{y} = f_\theta(x) = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

This implies that the minimal achievable loss is bounded below by a nonzero quantity due to the inherent ambiguity in the dataset.

More formally, since f_θ is a function and the dataset D does not represent a function (as it maps a single input x to multiple outputs y_i), it follows that f_θ cannot perfectly fit all target values simultaneously. Thus, the best the model can do under MSE is to output a central value—specifically, the mean of all y_i—which minimizes the squared error.

We now aim to formally show that this averaging behavior indeed arises by minimizing the following objective:

$$L(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2.$$

Since all input values x are equal, the corresponding predictions $\hat{y}_i = f_\theta(x) = \hat{y}$ are also equal. We therefore write:

$$L(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y_i)^2.$$

To find the minimum, we take the gradient with respect to $\hat{y}$ and set it to zero:

$$\nabla_{\hat{y}} L(\hat{y}, y) = \frac{2}{n}\sum_{i=1}^{n}(\hat{y} - y_i) = 0 \quad \Rightarrow \quad \sum_{i=1}^{n}(\hat{y} - y_i) = 0 \quad \Rightarrow \quad n\hat{y} = \sum_{i=1}^{n} y_i \quad \Rightarrow \quad \hat{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

Thus, the MSE loss is minimized when the network output $\hat{y}$ equals the mean of all target values.

Now we compute the corresponding minimum value of the loss by inserting $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ into the loss function:

$$\begin{aligned} L(\bar{y}, y) &= \frac{1}{n}\sum_{i=1}^{n}(\bar{y} - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(\bar{y}^2 - 2\bar{y}y_i + y_i^2) = \frac{1}{n}\left(n\bar{y}^2 - 2\bar{y}\sum_{i=1}^{n} y_i + \sum_{i=1}^{n} y_i^2\right) \\ &= \bar{y}^2 - 2\bar{y}^2 + \frac{1}{n}\sum_{i=1}^{n} y_i^2 = \frac{1}{n}\sum_{i=1}^{n} y_i^2 - \bar{y}^2. \end{aligned}$$

Therefore, the minimum achievable loss is given by:

$$L(\bar{y}, y) = \frac{1}{n}\sum_{i=1}^{n} y_i^2 - \left(\frac{1}{n}\sum_{i=1}^{n} y_i\right)^2,$$

which is the *empirical variance* of the y_i values. Since the dataset is ambiguous (i.e., not all y_i are equal), the variance is strictly positive:

$$L(\bar{y}, y) > 0.$$

This proves that a neural network trained using MSE on ambiguous data will output the mean of the targets and cannot reduce the loss arbitrarily—highlighting a key limitation in such settings.

It is now evident that in Equation 3, the loss $L(\hat{\mathbf{y}})$ satisfies $L(\hat{\mathbf{y}}) = 0$ if and only if Equation 1 is fulfilled. Since the loss function is always non-negative, this condition also corresponds to a global minimum.

An optimization algorithm such as Adam will naturally converge to one of the minima of the loss landscape. In the presence of multiple local minima—arising, for instance, from ambiguous solutions—the optimizer will simply settle on one of them.

Therefore, ambiguities are inherently handled by our approach: instead of averaging across all possible outputs (as in the supervised case), the network trained using the residual loss from Equation 3 will converge to a valid solution that satisfies the residual equation, avoiding the undesirable averaging effect.

D Continuous output space

```julia
function assign_to_cluster(point, centers)
    min_dist = 1e5
    min_ind = 0
    i = 1
    for center in eachcol(centers)
        dist = norm(point .- center)
        if dist < min_dist
            min_dist = dist
            min_ind = i
        end
        i += 1
    end
    return min_ind
end


y0 = [0., 1.4] # start value of simulation
cluster_ind = assign_to_cluster(y0, cluster_centers)
outs = []
for i in 1:size(inps)[2]
    yi = model_list[cluster_ind](inps[:,i]) # only evaluate one of multiple models
    push!(outs, yi)
    cluster_ind = assign_to_cluster(yi, cluster_centers) # find new cluster
end
println(outs)
```

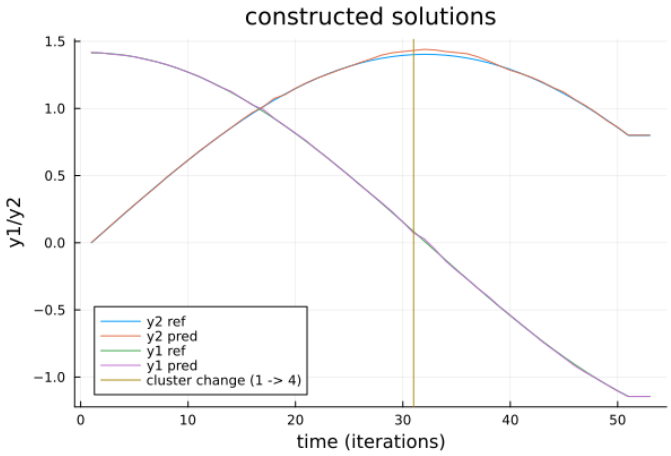

Figure 15: Constructed solutions for both output variables in the complex square root example. The vertical line marks the transition from cluster 1 to cluster 4 during simulation; after this point, the model trained on cluster 4 is used for prediction.

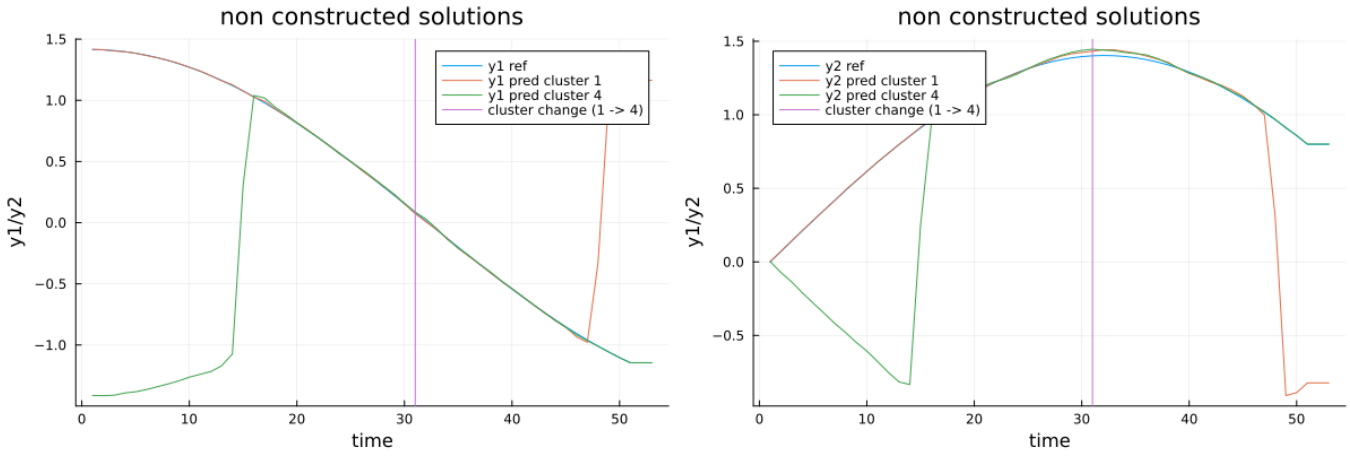

Figure 16: Non-constructed solutions for the output variables y_1, y_2. At the beginning of the simulation, the model trained on cluster 4 does not follow the reference trajectory; toward the end, the model trained on cluster 1 fails to do so. This illustrates that using either model alone would be insufficient, highlighting the utility of the switching algorithm.