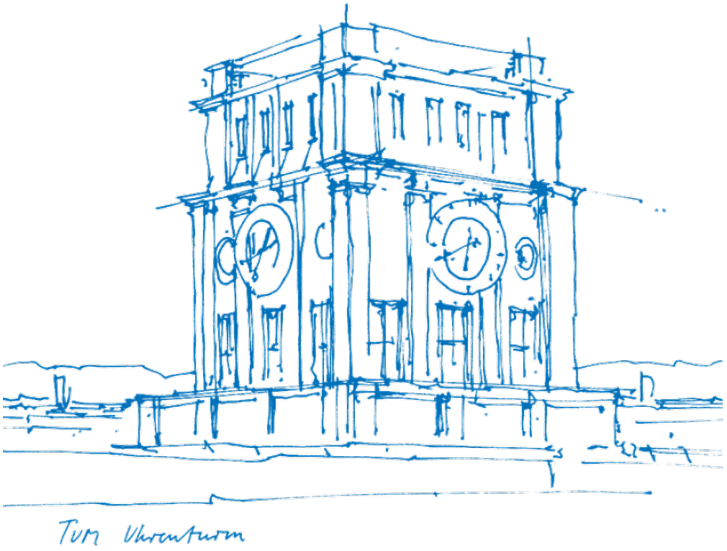
Chair for Scientific Computing
School of Computation, Information and Technology
Technical University of Munich

Coupling physics-informed neural networks from Nvidia Modulus with conventional models using preCICE

Dawid Klimont



Coupling physics-informed neural networks from Nvidia Modulus with conventional models using preCICE

Dawid Klimont

Chair for Scientific Computing
School of Computation, Information and Technology
Technical University of Munich

Coupling physics-informed neural networks from Nvidia Modulus with conventional models using preCICE

Dawid Klimont

Thesis for the attainment of the academic degree

Bachelor of Science (B.Sc.)

at the School of Computation, Information and Technology of the Technical University of Munich.

Examiner:
Prof. Dr. Hans-Joachim Bungartz

Supervisor:
Benjamin Rodenberg

Submitted:
Munich, 31.03.2025

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 31.03.2025

Dawid Klimont

Abstract

Recent advances in artificial intelligence and machine learning have led to the development of many tools for implementing and using neural networks. One field of interest is physics simulation, where systems governed by Partial Differential Equations (PDE) and Ordinary Differential Equations (ODE) are solved and approximated by conventional numerical solutions. A modern approach to solving such systems includes using machine-learned Neural Networks (NN) trained on recorded data and known governing equations of a system. The resulting NN is a Physics-Informed Neural Networks (PINN) and can be used as an efficient surrogate solver for the original system. One such software is the open-source Python library Nvidia Modulus (recently renamed Nvidia PhysicsNeMo), enabling users to implement, execute, and train such PINNs. The following thesis investigates whether and how a modulus model could be implemented and coupled through the physics simulation coupling library preCICE to solve a simple partitioned heat problem. The result provides the groundwork for future implementations where parts of a multi-physics system do not have a reasonable or efficient known conventional solution and warrant such a surrogate solution.

Contents

1 Introduction

This thesis aims to use the open-source Python library Nvidia Modulus to answer if it is possible to couple a Neural Networks (NN) from Modulus through the coupling library preCICE [10]. The library Modulus enables the training and implementation of Physics-Informed Neural Networks (PINN). These PINNs are a recent advancement in the field of physics simulation [7]. They enable the training of Models directly on Partial Differential Equations (PDE) or Ordinary Differential Equations (ODE) constraints, even if none or only a limited amount of data is available. With that, a surrogate solution can be created even if no numerical solutions exist or when the existing ones are too inefficient for reasonable use. The coupled approach enables multi-physics problems where parts of the system have that exact issue, that is solveable by integrating a Modulus NN into the system.

The thesis aims to answer whether and how creating a model that can be coupled with a conventional Finite Element Method (FEM) solver in a specific setup to become a surrogate for the existing coupled conventional solver is feasible. The insights gained could become the groundwork for analyzing and implementing Modulus-based neural networks through the coupling library preCICE.

Similar work exists mainly in using conventional solvers to generate data sets to train a NN like in [12], where smoothed FEM is used to prepare data for PINN training. However, this thesis will use coupling during training. Through bi-directional coupling, a FEM solver and the PINN will attempt to solve the same problem on different domains that share a boundary. With this, the feasibility of the software can be assessed, and a preliminary analysis of potential issues can be done when training PINNs in such a coupled way.

This thesis structure is as follows. First, the chapter 2 will give a rough overview of the mathematical fundamentals for the software used in the later implementation. Then chapter 3 will showcase the relevant software used to create the project. In chapter 4, the project code will be shown and explained. Then, in chapter 5, the project outputs and training behavior are analyzed. Finally, in chapter 6, the insights found and open answers for future work will be discussed.

2 Fundamentals

This chapter will list and explain all the fundamental background needed for further chapters. First, in section 2.1, the basics used for PDEs are explained, and then section 2.2 will explain the math behind the coupled conventional solver. Next, section section 2.3 will introduce core PINN concepts. The section 2.4 will explain the coupling scheme used for the later experiments.

2.1 Partial differential equations

PDEs and ODEs are commonly used to describe and compute solutions for various physics systems. They typically consist of a domain the solution is sought on, boundary conditions defined on the domain's edges, and interior constraints defined by differential equations. For transient problems, an initial condition is also required to get a solution. The resulting solution has then to satisfy these constraints.

The problem worked on during this thesis is a transient heat problem with a simple two-dimensional geometry. This is chosen explicitly to make validation of the functionality of later software easier. A so-called manufactured solution is used to provide the exact solution in advance. This makes the validation process straightforward. Usually, that exact solution is the unknown sought-after function, and solvers are used to compute it. However, our project uses it to validate the given solvers to have a foundation for their correctness. We define the PDE as follows:

$$\frac{\partial u}{\partial t} = \triangle u + f \quad in \quad \Omega \times (0, T]$$

$$u = u_m \quad in \quad \partial\Omega \times (0, T]$$

$$u = u_m \quad in \quad t = 0$$

With u_m being the known manufactured solution for a specific source term f. Any solver must approximate this solution on the entire domain and time interval. The first equation represents the governing PDE and so-called interior constraint. The second enforces the boundary condition, and the third specifies the initial condition. To keep the problem simple, a unit square is chosen as the given geometry of the domain. Finally, the chosen manufactured solution is defined.

$$f = \beta - 2 - 2\alpha$$

$$u_m = 1 + x^2 + \alpha y^2 + \beta t$$

This formulation can be used in later chapters to focus on if a solver approximates this solution for arbitrary given α and β

2.2 Example conventional solution the finite element method

Many approaches exist to numerically solve or approximate a PDE. This section will focus on the solution of the FEM. This method was chosen to be represented since the participant's conventional solver for the experiments will be using FEM.

The FEM approach discretizes the PDE domain into small elements or meshes. Then, each singular element approximates the solution for their local domain part. Finally, the entire solution can be constructed

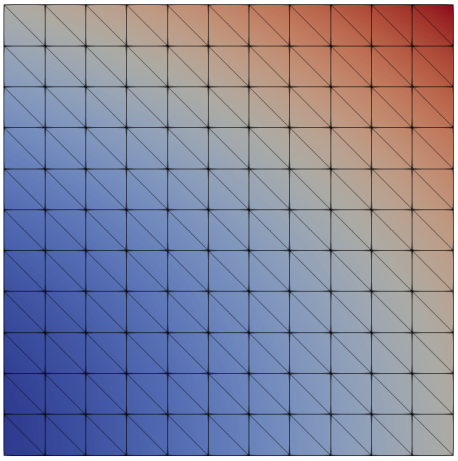


Figure 2.1 An example mesh over the previously defined domain

by combining all the elements and functions. This approach enables the numerical solution of even complex problems and geometries if the problematic areas are subdivided into sufficiently small, hence finite elements.

On top of that, this approach does not rely on finding a solution through the strong solution. Where parts of the equation require the second derivative of the sought-for function. Like for example, the Poisson equation:

$$\triangle u = f$$
$$\triangle u = \frac{\delta u}{\delta x^2} + \frac{\delta u}{\delta y^2}$$

Instead, it uses the weak solution to calculate local values using test functions. Then, with local approximations, a function like the hat function can be scaled to satisfy the equation for a given element.

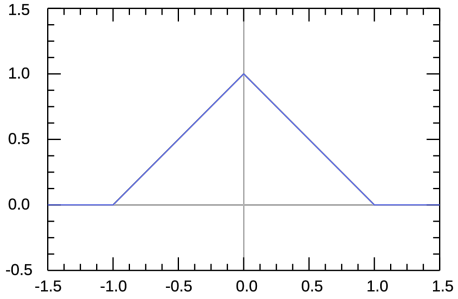


Figure 2.2 An example of a hat function plot `https://en.wikipedia.org/wiki/Triangular_function`

The weak solution then looks the following:

$$\int_\Omega (\triangle u \cdot v) d\Omega = \int_\Omega (f \cdot v) d\Omega$$
$$\int_\Omega (\nabla u \cdot \nabla v) d\Omega = \int_\Omega (f \cdot v) d\Omega$$
$$\nabla u = \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}$$

This form uses test function v to determine specific values for the finite elements solution. With each element then approximated, the entire solution can be solved by simply aggregating all the elements' functions, resulting in a valid solution for the problem domain without potentially satisfying the strong solution. For further reading and a detailed explanation, see [3].

2.3 Physics-informed neural network approach

The 1988 released paper [1] already determined that it is possible to use and adjust a Feedforward Neural Network (FFNN) to approximate any measurable function. Later, the optimization method Adam [4] was introduced, improving convergence behavior and stability for training NNs. With this, the foundations were created for the modern-day innovation surrounding the advances in machine learning. In order to have a better grasp of what specific parts of the Modulus implementation do, this section will give a brief overview of the key concepts of how machine learning a neural network works and how the innovative approach of PINN expands upon it found in [7].

Let us first define a NN. For this thesis, the Fully Connected Neural Network (FCNN) will be focused on. The FCNN is a subset of the FFNN. The FCNN name results from each neuron/node inside the network being connected to every node in the previous and subsequent layer. They consist of an input layer where the data can be inserted and an output layer where the function results can be extracted. This process of calculating a neural network response is called forward propagation. It is possible since the networks are feedforward only, which means results only get passed toward the last layer with no internal loops for calculations. On top of that, the main calculations of this network are done by so-called hidden layers between the output and input layers. These layers have all the exact predefined neuron count and must be evaluated during each forward propagation.

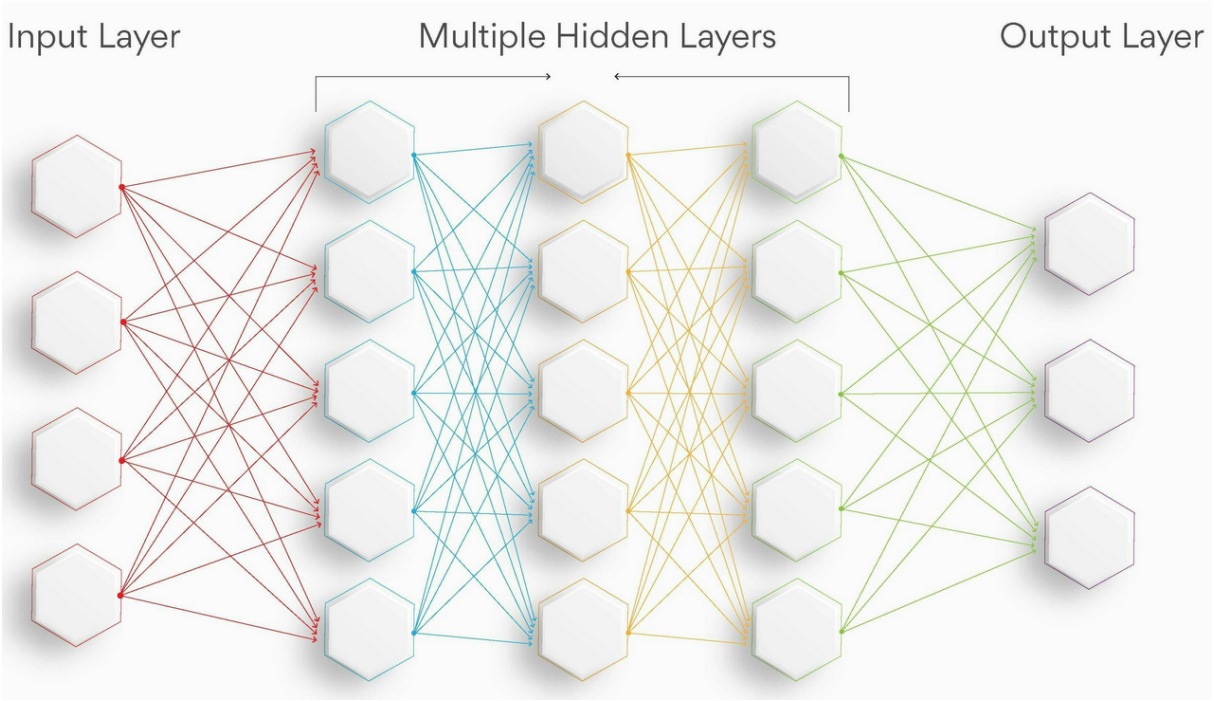


Figure 2.3 An example FCNN with three hidden layers with five neurons `https://www.turing.com/kb/importance-of-artificial-neural-networks-in-artificial-intelligence`

The neurons inside all of the layers operate on the same rules. To understand their operation, we will focus on a single neuron. Each neuron first does a weighted linear combination of all nodes in the previous layer. Each neuron from a previous layer provides its result. The weights for each connection are initially randomly selected. Depending on the model architecture, an internal bias value is also added, which is initially randomly selected. We hand that sum off to the chosen activation function of the network. This function then modifies our result, and we can pass that value off to all the nodes in the next layer until we reach the last layer, where the model's output is the output of the nodes.

$$activation_i^l = \sigma(bias_i^l + \sum_j \theta_{ij}^l \cdot activation_j^{l-1})$$

This equation shows that the weights and biases are important in what function is being approximated. These values are initially randomly chosen and later refined with backward propagation. It is important that all edges in the model have their weight and that each node has its own bias.

The model calculation is just a chain of linear combinations and an activation function. This allows us to use automatic differentiation later since the linear combination can be differentiated. If we choose an activation function with the same property, we will get an entire model capable of being differentiated. While convoluted when done by hand, this process is straightforward for a computer-based algorithm.

Many different activation functions are used in NNs, each with distinct advantages. The chosen activation function in this thesis project is the Hyperbolic Tangent (Tanh) function.

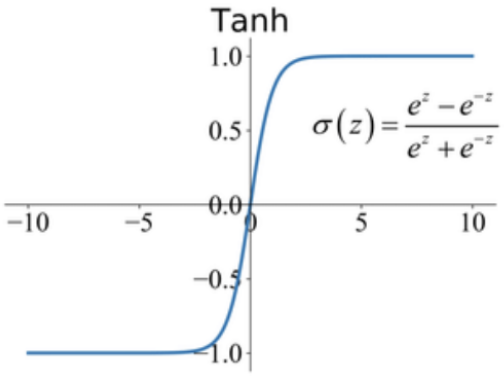


Figure 2.4 The Tanh function `https://paperswithcode.com/method/tanh-activation`

Now, if we use forward propagation to get an answer from a model for a given input, we can use that value to define a loss compared to the expected result using the to approximate function. This equation can then be differentiated with regard to any parameter in the model. With that, we can determine how a specific parameter has to be changed to reduce the loss of the model to the function.

$$L_u = || \; u_{mod}(input) - u_{exp} \; ||$$

The loss/performance function is commonly more sophisticated, with a mean squared error applied to multiple inputs. Then, back propagation uses the chain rule recursively to compute the loss function gradients with regard to each weight and bias in the model by starting in the last layer and propagating through each layer, updating all the parameters correspondingly.

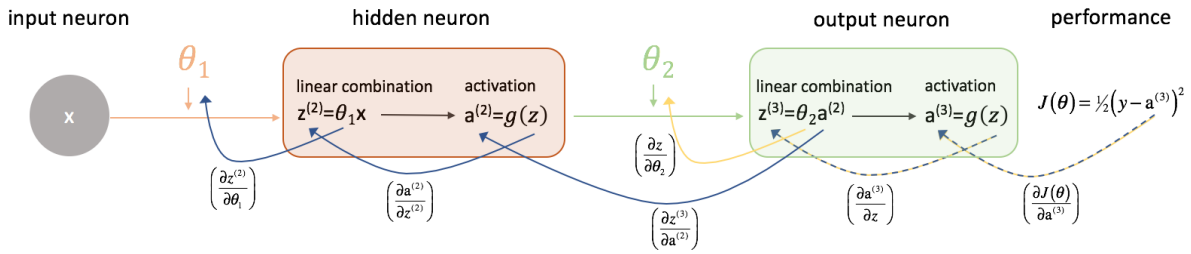


Figure 2.5 A one layer with one node FCNN back propagation `https://www.jeremyjordan.me/neural-networks-training/`

Then, each parameter gets subtracted from the multiplicative of the learning rate with the corresponding derivative.

$$\theta_{new} = \theta_{old} - lr_{cur} \cdot \frac{\partial L_u}{\partial \theta_{old}}$$

The learning rate is usually a function that decays with each backward propagation. The later implementation uses, for example, an exponential learning rate decay, where

$$lr_{cur} = lr_s \cdot decay^{steps}$$

With this knowledge, we can train a model on existing data or equations that we can evaluate. This would enable a data-driven approach to approximate a physical system by providing this training process with enough data points. It can also enforce simple constraints from a PDE, like boundary conditions, where we can evaluate the expected value to generate the loss function. However, the model must fulfill the differential equation for physics-driven machine learning. It is not straightforward, though, to get u_{exp} from a differential equation constraint like the interior constraint from section 2.1. However, the innovative solution in [7] is that we can also apply automatic differentiation with regard to the input neurons since models consist of differentiable operations. This means that when defining a PDE loss, the loss function uses the model as the sought-after solution and attempts to fulfill the governing constraint using its derivatives. Which in turn can be used to perform a loss function. That loss function can then be used in backward propagation like before. An example loss function for the section 2.1 found PDE:

$$L_{res} = || \frac{\partial u_{mod}}{\partial t} - \frac{\partial^2 u_{mod}}{\partial x^2} - \frac{\partial^2 u_{mod}}{\partial y^2} - f ||$$

This is usually calculated as a mean squared error over multiple inputs in a single training step. However, the exact implementation of all the automatic differentiation is handled in Modulus by PyTorch [6] and is only needed to understand how the model attempts to satisfy the governing constraints. For more reading on current PINN development and state [14].

2.4 Dirichlet Neumann coupling

As mentioned, this thesis will focus on a coupled setup of PDEs. The coupling algorithm will be the Dirichlet-Neumann approach [2]. With this, we split a domain into two separate ones connected via a coupling boundary. We define one solver as the Dirichlet solver and the other as the Neumann solver. These domains' only difference from 2.1 is that the Neumann solver receives the flux instead of a temperature on Γ_D from Figure 2.6. In addition, the initial solution requires the Dirichlet Solver to assume a starting value on Γ_D. We then can iterate over both of them repeatedly, starting with the Dirichlet solver. Then, the coupled boundary condition for the other solver is updated with each iteration using the current solution with a given coupling scheme. This process can then iterate upon these solvers until a specified convergence.

The existing solvers for this problem are split into two 1x1 rectangular Domains as seen in Figure 2.6, with the Neumann solver being translated in the x direction by 1. This leads to a coupling boundary Γ on x=1. In the case of this thesis, the aim is to determine how to create a NN coupled through preCICE as the Dirichlet part of such a setup and analyze its behavior in such a coupling algorithm.

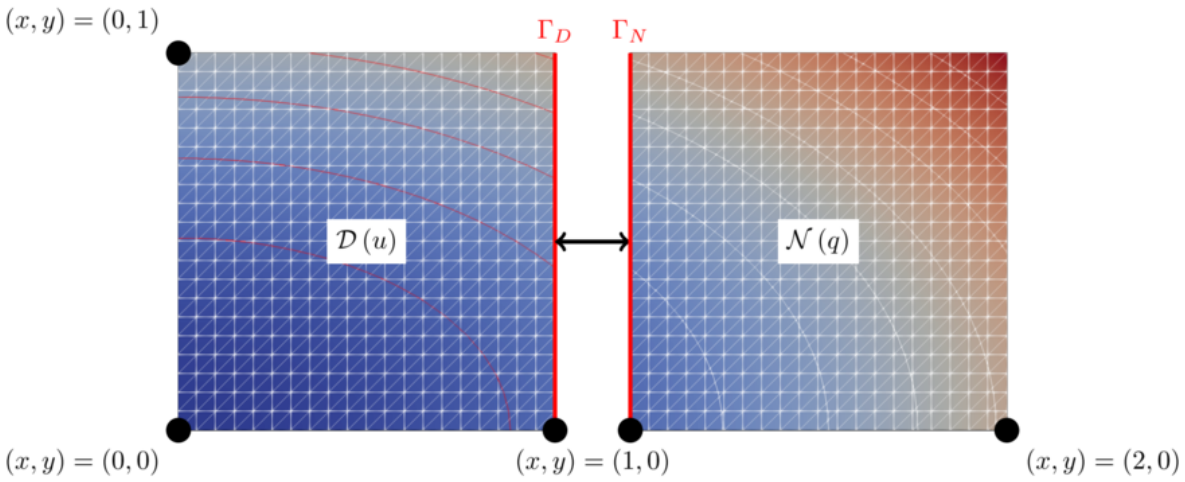


Figure 2.6 The Dirichlet Neuman partitioning of the target problem

3 Software

This chapter presents all of the software that was mainly used for this thesis project. The first showcased software in section 3.1 is the previously mentioned PINN library Nvidia Modulus. After that, the section 3.2 will showcase the preCICE library used as the coupling software. Finally, in section 3.3, the FEM solver FEniCS will be introduced with its corresponding adapter in preCICE.

3.1 Modulus

Modulus is an open-source Python library developed by Nvidia for using and training PINNs. Its libraries are split into two separate repositories. The first is Modulus Core, which implements the ideas from section 2.3 by using and expanding on PyTorch. The other is Modulus Sym, a wrapper for the Core library that aims to simplify the machine learning implementation and make project code more Pythonic. The Modulus Sym library enables users to train neural networks with just a good understanding of the problem domain. Both libraries allow the training of models using data-driven, physics-driven, or mixed approaches. In the projects in chapter 4, most components used are from Modulus Sym. The Libraries provide diverse tools to implement a neural network from many different architectures or use built-in implementations like activation functions or optimizers. The following Figure 3.1 shows the structure of a Modulus Sym model implementation. As seen in the graphic, the first step of any Modulus Sym implementation is loading the

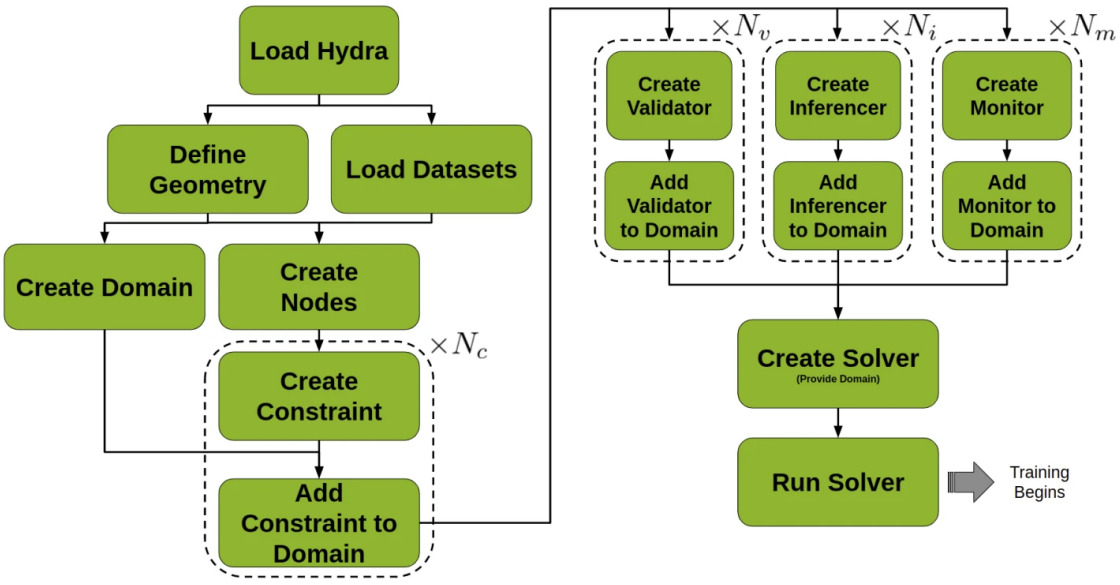


Figure 3.1 The intended Nvidia Modulus Sym workflow `https://docs.nvidia.com/deeplearning/physicsnemo/physicsnemo-sym/user_guide/basics/physicsnemo_overview.html`

configuration through an included library called Hydra. Then, either the physics-driven approach is chosen by creating a geometry or the data-driven approach by loading the data. A mixed approach there would require both. Then, a domain object is created, which receives all the constraints on which to train the model. These can be simple data constraints from the loaded data, equations that generate direct values like boundary conditions, or complex interior constraints with a differential equation. Then, any number

of validators, inferences, and monitors can be added to extract and monitor the model. Then, finally, the solver gets created and starts the actual machine learning process.

3.2 preCICE

The open-source coupling library preCICE is a software package for multi-physics problem solutions where multiple different solvers have to interact with each other. The core idea is that with the APIs provided by preCICE, the coupling implementation can be done without knowing about the other participant's software. This black-box approach aims to increase modularity to simulation setups, leading to far more manageable maintenance and development of systems.

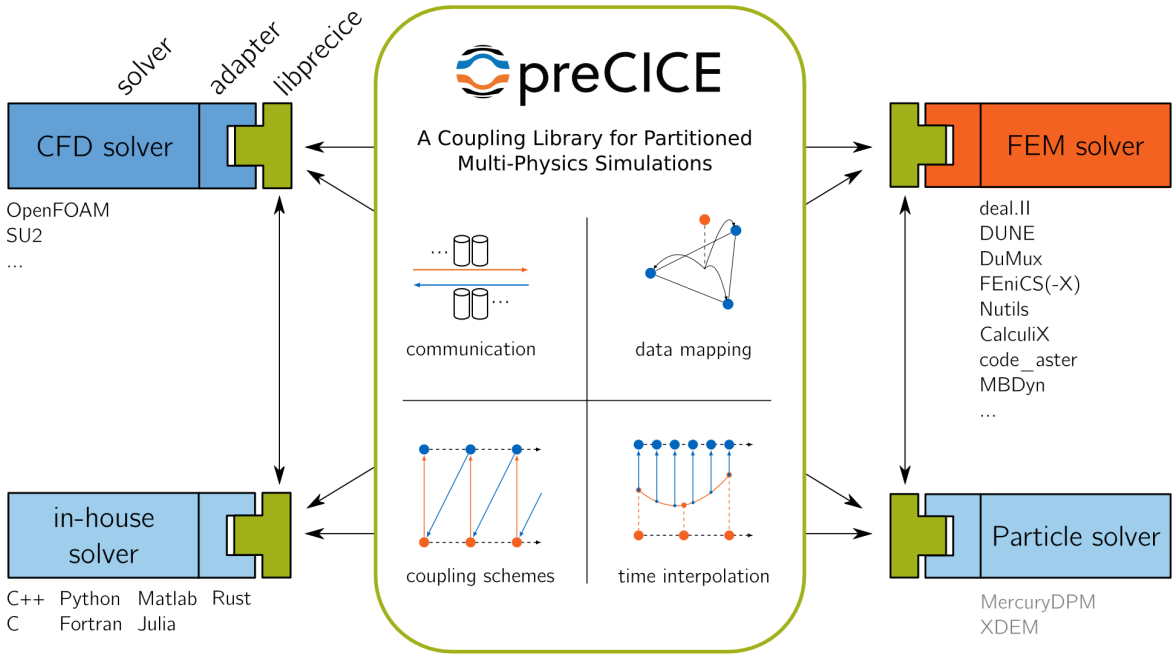


Figure 3.2 The preCICE components [10]

Figure 3.2 shows all the parts included by preCICE [10]. Using the preCICE configuration file, many settings and options can be modified and chosen. preCICE lets us define any solvers as participants with read and write objects. Additionally, coupling schemes need to be determined for a coupled simulation. Supported schemes have to either be serial, meaning each solver gets executed sequentially, or parallel, where the solver calculates at the same time. However, parallel execution is not applicable for the thesis use case since the model training will take substantially longer than the conventional solver for this problem. Another part of the coupling scheme is whether it is explicit or implicit. These determine if a time frame is only iterated once or multiple times. Time frames are defined subsections of the whole time range determined in the configuration. Smaller timer ranges can improve conventional solver accuracy. preCICE also handles time and data mapping for the simulation between the solvers so that they can internally use their substeps and meshes with no regard to what the partner uses.

Another key aspect of the preCICE usage is how the data is transferred. All participants must use a coupling expression capable of being updated with new data at any time step. On top of that, how the data is propagated is also relevant. Supported is a simple under-relaxation, where preCICE merges the solution from the previous iteration weighted by the configurated value with the new one to get the solution it actually propagates to more complex once as found in [9].

3.3 FEniCS and the FEniCS preCICE adapter

As mentioned in section 2.2, the other coupled solver for the implementation will be the conventional numerical FEM solver FEniCS. The coupled partitioned heat tutorial already employs this open-source library for solving PDEs. From the library itself, this thesis will only use a limited amount of functionalities since the aim is to quickly build a prototype using the existing implementation and dedicated adapter to prove if the primary goal of this thesis is attainable. Only the necessary parts to initiate and use the adapter and FEniCS expressions for the Modulus constraints will be used in the code examples. For a more detailed look into FEniCS, see [5]. The dedicated FEniCS adapter [8] builds on top of the preCICE API to provide easy coupling using FEniCS-based meshes and functions. It can be seen as the glue code in between preCICE and in FEniCS usable objects.

4 Implementation

The following chapter will first include in section 4.1 a summary of all the necessary dependencies to run the code, including conda installation steps to replicate a similar environment. In section 4.2, the code for a transient two-dimensional heat equation model will be shown and elaborated upon. Each part of the code is dissected in correspondence to the summary in section 3.1. Then, in section 4.3, all the software functionalities needed for the coupled implementation will be listed and shown how they can be implemented. Finally, in section 4.4, an overview of the coupled implementation will be given.

4.1 Requirements

Concrete preliminary steps needed to run and further develop the code created during this thesis will be listed. Essentially, necessary hardware requirements and how to create a Python environment capable of executing the created scripts and using the Modulus models will be explained.

4.1.1 Hardware

To be able to run Nvidia Modulus, the system requires a supported Nvidia GPU[1]. This requirement exists since Modulus uses PyTorch to manage and train models. While PyTorch has a CPU-bound version and supports CPU tensor calculation on all versions, Modulus uses GPU-bound tensors. On top of that, Modulus requires a GPU capable of using recent CUDA versions and limiting the useable GPUs to the ones mentioned by Nvidia or newer. For the creation of this thesis, an RTX 3080 GPU and an AMD EPYC 7402 CPU were used with Ubuntu 20.04.2.

Modulus also supports the exporting and importing NNs to and from PyTorch, enabling the ability to run a Modulus-trained model as a pure PyTorch program, enabling it to run the operations solely through the CPU. However, this thesis will not use this since the main goal is to determine what the coupling of Modulus-based software could be implemented.

4.1.2 Dependency management

It is possible to install Modulus through a Docker file[2]. However, for this thesis, we will implement the code inside a conda environment. In order to enable the replication of the results and provide a basis for future work on similar setups, the required installation instructions for the Python environment are now summarized. In order to manage the dependencies, the open source package management system conda is utilized. The following installation commands were used to make this thesis.

```
conda create -n modulus_env python=3.10
conda activate modulus_env

conda install -c pytorch -c nvidia -c conda-forge fenics=2019.1.0 pyprecice=3.1.2   \
    fenicsprecice=2.2.0 pytorch=2.5.1=py3.10_cuda12.1_cudnn9.1.0_0                  \
    torchvision=0.20.1=py310_cu121
```

[1] https://nvdam.widen.net/s/gfcwrp7mq2/hpc-for-dev-modulus-datasheet
[2] https://catalog.ngc.nvidia.com/orgs/nvidia/teams/modulus/containers/modulus

```
pip3 install nvidia-modulus==0.9.0 nvidia-modulus.sym==1.8.0 torch==2.5.1          \
    torchvision==0.20.1
```

The conda installer allows for the installation of software and compilers on machines where the user would need administrative rights to install corresponding components on the system side. Since Nvidia has stopped supporting the conda packages for Modulus, this installation resolves this by installing core dependencies through conda and then enforcing the use of the installed PyTorch version in the pip install. With this, a Python environment can run the code examples and load the resulting models.

4.2 Two dimensional transient heat equation neural network

This section includes the basic implementation of a Model solving the in section 2.1 mentioned transient heat PDE. The aim is to split the code into easily digestible chunks, which provide a comprehensive tutorial for anyone intending to replicate the result or expand on this software. The complete code can be found at [3].

4.2.1 Configuration

First, the configuration file for storing the model setting will be examined. The following conf.yaml file is being stored in the local conf folder for this project. Every Modulus project uses this style of Yaml file to store hyperparameters used for training and generating many objects. However, this thesis will focus on the bare minimum configuration since most hyperparameters can also be implemented inside code, making the code easier to follow.

```
1  defaults:
2      - modulus_default
3      - scheduler: tf_exponential_lr
4      - optimizer: adam
5      - loss: sum
6      - _self_
7  scheduler:
8      decay_rate: 0.90
9      decay_steps: 2_000
10 training:
11     rec_results_freq : 1_000
12     save_network_freq : 1_000
13     rec_validation_freq : 10_000
14     max_steps : 100_000
15     summary_freq : 100
```

The defaults section includes all the necessary settings to generate and run a modulus model. Of importance are the choices of scheduler and optimizer. In our example, we will use the Adam optimizer [4] and an exponential learning rate decay. The scheduler part defines with the scheduler setting in defaults how a model's learning rate will decline during training. The used exponential decay is defined in section 2.3 The initial learning rate can also be set in addition to many other settings. However, these will be used in their default state and can be read in the corresponding documentation[4]. Finally, to change the frequency of certain operations in the training settings, line 12 defines the step count to save the model as a checkpoint.

[3] https://github.com/DawidKlimont/tutorials/tree/develop/partitioned-heat-conduction/solver-modulus/basic_model

[4] https://docs.nvidia.com/deeplearning/modulus/modulus-v2209/user_guide/features/configuration.html

Here, it is specified that a checkpoint be created every 1000 steps. This also includes step 0. In addition, line 13 declares that any validators created shall be applied every 10000 steps. Finally, the target step count is defined in line 14, at which Modulus will stop training the model. Of note are line 11, which sets any other not included freq to itself, and line 15, which determines logging frequency.

4.2.2 Code structure

The following chapters explain the uncoupled project code. It can be run by calling python heat.py, python3 heat.py or running the local script with ./run.sh. Next, the core structure of the code will be defined through the run function.

```
1  from modulus.sym.hydra import ModulusConfig
2
3  @modulus.sym.main(config_path="conf", config_name="config")
4  def run(cfg: ModulusConfig):
5      alpha, beta, scaling = 3.0, 1.2, 10.0
6
7      u_net = initialize_neural_network()
8      nodes, geometry = initialize_nodes_and_geometry(u_net, alpha, beta, scaling)
9      constraints = initialize_constraints(nodes, geometry, alpha, beta, scaling)
10     validator = initialize_validator(nodes, alpha, beta, scaling)
11     domain = initialize_domain(constraints, validator)
12     train_model(domain, cfg)
13
14 run()
```

Here, the previously defined configuration file is loaded via the decorator. Thus, the decorator gets handed the path to the name of the file. Modulus recommends using this decorator to initialize the configuration object. It is possible to create and load the configuration without it, using the hydra function compose found in the Modulus Sym library. However, this solution is not recommended and is referred to by comments in the corresponding files as a last-ditch effort and unnecessary for the coupling implementation.

First, the problem parameters are set inside the run function, after which every relevant implementation step has been refactored into its function. A new addition is the scaling variable, which will be used to rescale the problem internally onto the result range of 0 to 1, which is recommended by Nvidia[5] in order to improve convergence behavior of the NN since many activation function work best in that range.

With this, each of the following chapters corresponds to each of the functions. Line 7 will be defined in subsection 4.2.3. Then, line 8 will be explained by subsection 4.2.4. Next, line 9 will be elaborated by subsection 4.2.5. Then, line 10 will be skipped and examined later by subsection 4.2.7. Finally, the Training from line 12 and additionally line 11 will be described by subsection 4.2.6

4.2.3 Initiating the neural network

Now, how the neural network is initialized will be described. The key settings will also be included in detail. These settings, like the chosen activation function, model type, or model size, can also be defined in the configuration file. Depending on the project, the decision on where to store this can differ. In this case, these settings are in the Python script for easier development. However, significantly more extensive projects can become more structured when these parameters are stored in an organized configuration file.

[5] https://docs.nvidia.com/deeplearning/modulus/modulus-v2209/user_guide/theory/recommended_practices.html

```
from modulus.sym.key import Key
from modulus.sym.models.activation import Activation
from modulus.sym.models.fully_connected import FullyConnectedArch

def initialize_neural_network():
    u_net = FullyConnectedArch(
        input_keys = [Key("x"), Key("y"), Key("t")],
        output_keys = [Key("u")],
        activation_fn = Activation.TANH,
        layer_size = 128,
        nr_layers = 7,
    )
        return u_net
```

The neural network architecture is defined by the imported FullyConnectedArch class. If an implementation with a variable architecture is required, the Modulus function instantiate_arch provides a solution that can determine its architecture during runtime. However, the FullyConnectedArch class is suitable for this project's purpose. The FCNN was described in section 2.3.

Lines 7 and 8 define the model's input and output layers. The spatial coordinates x, y, and the time t are defined as the input keys. The resulting output layer of the model, consisting of the sought value temperature u, is defined by the output keys.

Line 9 defines the chosen activation function for the model's neurons. Here, we apply the Tanh function using Activation.TANH, which was described in section 2.3. If the model does not receive a declared activation function, it will default to the Sigmoid Linear Unit Function. Different activation functions can significantly alter the model's behavior and learning. How the Tanh function compares to the multitude of other functions provided by modulus, especially in this setup, is another potential question for further work.

Finally, we define the model's internal properties. The hidden layer count is chosen by nr_layers, and the neuron count of each layer by layer_size. The model's size of 7 layers with every 128 neurons was chosen based on the in [11] found the conclusion that a Modulus FCNN with six layers with 64 neuron model is best fitted for a 2-dimensional steady state heat transfer model. Since our implementation increases complexity by adding additional dimensions of time, a bigger model was chosen. Determining which size would fit this problem configuration best will not be part of this thesis since the focus lies on the feasibility of a coupled simulation, not its optimization.

4.2.4 Initiating the partial differential equation, geometry, variables and nodes

Now, we define the domain area and the governing equations. Modulus has pre-built classes for all kinds of common PDE and ODE problems, where only key parameters must be defined. It is, however, also possible to create completely custom equations. The second approach showcases how any arbitrary equation could be defined, which is why it was chosen for this implementation. For geometry, the modulus presents a large number of pre-built primitives. In addition to that, the library supports the loading of pre-built STL files. Since our domain is a square, the primitive Rectangle will suffice. The nodes created here are a Modulus wrapper of PyTorch node objects. These contain the necessary information for the model's training process and are used to generate an execution graph during training. In this case, the nodes will be generated from the neural network and the initialized equation object.

```
from sympy import Symbol, Function
from modulus.sym.eq.pde import PDE
from modulus.sym.geometry.primitives_2d import Rectangle

class HeatEquation2D(PDE):
```

```
6       def __init__(self, alpha, beta, scaling):
7           x,y,t = Symbol("x"), Symbol("y"), Symbol("t")
8           input_variables = {"x": x, "y": y, "t": t}
9           u = Function("u")(*input_variables)
10          self.equations = {}
11          self.equations["heat_equation"] = u.diff(t) -(u.diff(x,2)+u.diff(y,2))
12                                                      -(beta-2-2*alpha)/scaling
13
14  def initialize_nodes_and_geometry(u_net, alpha, beta, scaling):
15      geometry = Rectangle((0,0),(1,1))
16      eq = HeatEquation2D(alpha, beta, scaling)
17      nodes = eq.make_nodes() + [u_net.make_node("u_network")]
18      return nodes, geometry
```

Lines 5-12 show how an Equation can be defined for Modulus, and line 17 then creates the aforementioned nodes. This implementation was based on the 1D Wave equation example[6]. In lines 11 and 12, we create a SymPy equation representing the previously defined PDE to the equation from section 2.1. The optimizer will later use this equation during training to minimize the loss as described in section 2.3.

4.2.5 Declaring constraints

The constraint objects in Modulus are employed to add conditions to the model that it will be trained to fulfill. The following are the implemented constraints based on the in section 2.1 defined PDE.

```
1   from modulus.sym.domain.constraint import PointwiseBoundaryConstraint,
2                                             PointwiseInteriorConstraint
3
4   def initialize_constraints(nodes, geometry, alpha, beta, scaling):
5       x,y,t = Symbol("x"), Symbol("y"), Symbol("t")
6       time_range = {t: (0.0,1.0)}
7
8       initial_condition = PointwiseInteriorConstraint(
9           nodes = nodes,
10          geometry = geometry,
11          outvar = {"u": (1 + x*x + alpha*y*y)/(scaling)},
12          batch_size = 1_000,
13          parameterization = {t: 0.0},
14          fixed_dataset = False
15      )
16
17      boundary_condition = PointwiseBoundaryConstraint(
18          [...]
19          outvar = {"u": (1 + x*x + alpha*y*y + beta*t)/(scaling)},
20          batch_size = 1_000,
21          parameterization = time_range,
22          fixed_dataset = False
23      )
24
25      interior_constraint = PointwiseInteriorConstraint(
```

[6] https://docs.nvidia.com/deeplearning/modulus/modulus-v2209/user_guide/foundational/1d_wave_equation.html

```
        [...]
        outvar = {"heat_equation": 0},
        batch_size = 10_000,
        parameterization = time_range,
        fixed_dataset = False
    )

    constraints = []
    constraints.append(initial_condition)
    constraints.append(boundary_condition)
    constraints.append(interior_constraint)
    return constraints
```

Every constraint receives a reference to the nodes and geometry. The geometry limits what points are going to be sampled for training. PointwiseInteriorConstraint will sample the entire geometry, and PointwiseBoundaryConstraint will only sample the border. With batch size, we define how many values will be used in each training step. The more important or complex the constraint's behavior, the bigger the recommended batch size. In our case, the initial condition is more straightforward than the other two since it lacks the time dimension from parameterized to t=0 in line 13. On the other hand, the boundary constraint has far less area to train upon than the interior constraint, which also has to enforce a PDE instead of a simple equation. With this, the relative batch size has to be chosen. The system performance should determine the magnitude of the batch size. Since massive batches could outgrow the memory of the GPU, they have to be loaded on in addition to simply slowing down each training step since more data has to be processed. The outvar definition in each constraint is implemented according to the manufactured solution; especially line 27 defines the previously implemented PDE in subsection 4.2.4 to equate to 0. The final setting chosen to be included in this implementation is the fixed_dataset. Suppose we set it to false; the Modulus solver will, during training, continuously generate new data for training, else Modulus will generate a batch per step in an epoch(default 1.000) at the start of training and iterate over the same data for each epoch if we do not include it. With that parameter set to False, the trainer will be able to generate new data for each epoch at their start, which leads to more diverse data and more reliable training since, depending on the batch size, the data could provide incomplete information for training.

4.2.6 Instantiating the domain and training the model

With all the necessary parts created, the training process can be started. The following part finishes the functionality required for the model to work.

```
from modulus.sym.domain import Domain
from modulus.sym.solver import Solver

def initialize_domain(constraints, validator):
    domain = Domain()
    for constraint in constraints:
        domain.add_constraint(constraint)
    domain.add_validator(validator)
    return domain

def train_model(domain, cfg):
    solver = Solver(cfg=cfg, domain=domain)
    solver.solve())
run()
```

The solver object executes the training loop inside of the solve function. It also manages all constraints and monitors, validators, or inferencer objects through the domain object. At the beginning of the training, the solver will also attempt to load any existing checkpoints. The configuration can define the location; in this case, the defaults are in place, leading to the outputs folder inside the project location. There, the solver will also save any checkpoints once it reaches a step count that is a multiplicative of save_model_freq.

4.2.7 Extracting data

Since the validation of the model regarding whether or not it approximates the manufactured solution is of key interest, the following paragraphs will describe what tools modulus presents to us. First, the key model parameters are regularly stored in event files inside the output directory. Using Tensorboard[7], we can visualize the default monitored values and any added custom monitors. The logged loss data is mainly interesting, as it gives insights into how accurately the network predicts the training data output for any logged training step. Another option to visualize the data is using a validator. The following listing shows a possible implementation.

```
1  from modulus.sym.domain.validator import PointwiseValidator
2  from modulus.sym.utils.io import ValidatorPlotter
3
4  class CustomPlotter(ValidatorPlotter):
5      def __call__(self, invar, true_outvar, pred_outvar):
6          invar_subset = {"x": invar["x"], "y": invar["y"]}
7          true_outvar["u"]=true_outvar["u"]*10
8          pred_outvar["u"]=pred_outvar["u"]*10
9          return super().__call__(invar_subset, true_outvar*10, pred_outvar*10)
10
11 def initialize_validator(nodes, alpha, beta, scaling):
12     c, t = 1000, 1.0
13     X, Y = torch.meshgrid(torch.linspace(0, 1, c), torch.linspace(0, 1, c), indexing="ij")
14     invar = {"x": X.reshape(-1, 1), "y": Y.reshape(-1, 1), "t": torch.ones(c*c, 1)*t}
15     outvar = {"u":
16         (1+invar["x"]*invar["x"]+alpha*invar["y"]*invar["y"]+beta*invar["t"])/scaling
17     }
18     validator = PointwiseValidator(
19         invar = invar,
20         true_outvar = outvar,
21         nodes = nodes,
22         batch_size = c*c,
23         plotter = CustomPlotter(),
24     )
25     return validator
```

With this, we generate in lines 12-17 input and expected output data for a grid of points based on the known solution. The validator then gets called during training at given intervals and stores the results as VTU files visualizable by Paraview. In addition to that, there is a plotter class in the library capable of generating matplot graphics for two-dimensional models. To use this component in this setup, the validator is set to receive only inputs at a fixed time, enabling us to expand the plotter to ignore the time input tensor (lines 4-9).

[7] https://www.tensorflow.org/tensorboard

4.3 Requirements for coupling

This section describes the key software components created and gathered necessary to enable the coupling of a modulus model through preCICE.

4.3.1 Flux calculation

As mentioned in section 2.4, the Dirichlet solver is supposed to write the temperature flux onto the coupling boundary, and the previously created model only has the temperature as an output. We could expand our model to include an additional flux output to satisfy this requirement. However, this solution would increase the model's complexity and worsen the convergence since the trainer has to adjust the neural network to satisfy two differential equations. A far better solution is to use the automatic differentiation provided by PyTorch[8]. Since Modulus internally uses PyTorch for all the neural network operations, we can differentiate any model output based on the input tensor used.

```
input_dict = {
    "x": torch.tensor([[1.0]], device="cuda", requires_grad=True),
    "y": torch.tensor([[1.0]], device="cuda", requires_grad=True),
    "t": torch.tensor([[1.0]], device="cuda", requires_grad=True)
}
result = u_net(input_dict)["u"]
result.backward(torch.tensor([[1.] for _ in result], device="cuda"))
return x.grad
```

This shows how if all input tensors enable requires_grad, all the gradients of the model output in regards to corresponding input variables can be quickly calculated by the backward function. With this, x.grad includes the sought-for gradient at the specified input values. This is possible since the enabled settings lets PyTorch keep track of the on the tensor applied operations and then do automatic differentiation.

4.3.2 Coupled boundary condition

In subsection 4.2.5, the default approach to define a constraint using a SymPy expression was shown. In the coupled setup, however, the data is provided through the coupling interface. In the case of the FEniCS preCICE adapter, the received values are a FEniCS expression. The following approaches have been found to implement that into the constraints.

```
1 outvar = {"u": lambda x, y, t: numpy.vectorize(some_fenics_expression(x,y,t))},
2 or
3 outvar = fenics_function_wrapper,
```

The key problem is that when Modulus generates a batch of data, it is generated at once by calling the corresponding outvar expression using randomly polled points in the form of tensors for each input variable with the length of the batch size. In line 1, an approach is shown to enable a single line for an FEniCS expression to process this type of data inside a lambda expression. With NumPy vectorize, the tensor data gets automatically processed even though the inside function can only process scalar values. A custom function can be called with the second approach in line 3. Handling the tensors and outputting the correctly shaped data befalls the function. Both approaches work and can be used to process an FEniCS function into a modulus constraint. However, this data generation type must be done via a CPU. This is significantly slower than having an expression directly for GPU-bound tensors or loading tensor data onto the GPU. Modulus also provides the constraint property of num_workers to speed this up. This can enable

[8] https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

to parallelize the data generation process. However, it is not applicable when using a lambda or FEniCS expression since Modulus requires the code to be serializable into a byte stream for parallelization, which is not possible here.

4.3.3 Applying updated constraints

Since each iteration, the coupling condition on the shared boundary is being updated, as described in section 2.4, an implementation capable of changing the constraints objects applied to the domain objects is necessary. The following code was determined to enable the changing of the pre-existing constraints.

```
domain.add_constraint(previous_condtion_to_drop)
domain.constraints.clear()
domain.add_constraint(updated_condition_to_add)
```

It is possible to clear all the previously known constraints inside the domain object because they are stored inside a local dictionary. It is also possible to re-initialize the domain object, resulting in the same effect. With these two approaches, implementing an every-iteration updating constraint is possible.

4.3.4 Splitting the boundary conditions

Previously, the conditions on the domain only applied to either the boundary or interior of the geometry. Additionally, parameterization was used to determine the time range in which the constraints were applied. However, with the coupled approach, it is necessary to differentiate the coupled boundary where the solution of the other solver applies and the remaining boundary where the solution is known. For this purpose, the following criteria parameters are applied to the corresponding boundary constraint.

```
#For coupled
criteria=(x>0.5) & (y<1.0) & (y>0.0),
#For remaining
criteria=x<1.0,
```

Modulus allows SymPy expressions to be used in these criteria parameters, which are applied to the point sampling. Here, line 2 defines the coupling boundary as the inner right half of the domain, which, then applied to the boundary constraint object, will only generate points on the edges that satisfy this condition. Line 4 is then applied to the remaining known boundary constraint.

4.3.5 Restarting training

Finally, the ability to restart and continue training is also required. As mentioned in subsection 4.2.1, once training is started, the solver will continue the training until max_steps are reached. It is possible to include an additional stopping criterion to stop training when, for example, a small enough loss is achieved. However, this would not let the model continue if it had previously stopped training because of reaching the step limit. The max_steps parameter must then be modified during run time like the code below.

```
solver.solve()
solver.max_steps+=1_000
solver.solve()
```

The shown code edits the processed max_steps data from the configuration file inside the solver object. This is a possible approach to extend the training. It is also important that the model is best saved once the total step count is achieved since Modulus always loads pre-existing models when starting the solve

function. This can be adjusted through the corresponding freq setting in the configuration. If the training does not end on the saving checkpoint step, the restarting of solve will lead to overwriting this data with a previous checkpoint if checkpointing is not disabled.

4.4 Coupling to preCICE

Now, the two-dimensional transient heat model is implemented as a coupled solver. The main goal is to provide a surrogate for the already existing implementation of the Dirichlet FEniCS solver found in [9]. For this approach, to reduce implementation work since the main interest is in a proof of concept and not an adequately developed adapter, the existing FEniCS partitioned heat solver implementation and adapter configuration found in [13] is used as basis. In the preCICE configuration, the time window size gets set to the max time window since our model is best trained on the entire time range else; the model would start learning new behavior for the last time window after already training for thousands of steps with a far decayed learning rate. On top of that, the learning in the Modulus config is also relaxed a lot compared to the uncoupled implementation since our coupling boundary differs, especially early on, not quite a lot from the function we want to approximate. Additionally, we will keep using the serial implicit coupling scheme with under-relaxation acceleration of 0.5 and a 100 iteration maximum, which we will reach during training since the NN does not reach close to the set convergence. The following is an excerpt of the actual implementation found in [10].

```python
@modulus.sym.main(config_path="conf", config_name="config")
def run(cfg: ModulusConfig):

    dt, t_coupling, n = 0.01, 0.0, 0
    alpha, beta, scaling = 3.0, 1.2, 10.0

    coupled_boundary_expressions = []
    u_net = FullyConnectedArch([...])
    u_net.to("cuda")
    modulus = ModulusHelper([...])
    modulus.train_model([...])

    while precice.is_coupling_ongoing():
        if precice.requires_writing_checkpoint():
            precice.store_checkpoint(dummy, t_coupling, n)

        read(dt)
        coupled_boundary_expressions.append([...])
        write(u_net, dt, t_coupling)

        if precice.requires_reading_checkpoint():
            modulus.train_model([...])
            _, t_coupling, n = precice.retrieve_checkpoint()
            coupled_boundary_expressions = coupled_boundary_expressions[:n]
        else:
            t_coupling += dt
            n += 1
```

[9] https://github.com/precice/tutorials/blob/855ce6159e0370827f60d1c02f14e75c9be3a4c5/partitioned-heat-conduction/solver-fenics/heat.py

[10] https://github.com/DawidKlimont/tutorials/tree/develop/partitioned-heat-conduction/solver-modulus/heat.py

```
28
29      precice.finalize()
30
31  def initialize_fenics_and_adapter():
32      [...]
33  precice, coupling_expression, dummy = initialize_fenics_and_adapter()
34  run()
```

We will turn off the timeframes for the Modulus solver experiments as mentioned in section 3.2. With this, we only have to iterate multiple times in a singular timeframe, making the model's training process more manageable. First, in lines 31-33, we initiate all the necessary components to use the FEniCS adapter. This includes initializing the FEniCS mesh and FEniCS adapter with all the requirements. While also creating the coupling_expression that will receive the updated FEniCS expression from the adapter used for training on the coupled boundary later. Then, inside of run as done in subsection 4.2.2, the configuration will be loaded via the decorator. Outside of this, the only Modulus feature in the primary run function is the network creation in line 8. After this, a ModulusHelper class is included in the project, which contains all of the Modulus operations. The Modulus Class will not be discussed in detail since it aggregates everything found in chapter 4 and can be found in the above-mentioned repository.

The important thing is that the init function creates all the necessary components, and the train model function trains the neural network for a thousand steps while taking an array of functions with timestamps to apply as boundary constraints on the coupled edge. In general, lines 1-12 initialize the solver with its settings and do the initial training to generate a guess for the first round of the Dirichlet-Neumann coupling. After that, in the while function, the code iterates over all the timesteps between 0 and 1.0 in 0.01 steps, and on each step, it reads the coupling boundary and stores the vectorized FEniCS expression in an array later used for the training. In addition, it also calls the write method, which extracts and writes the flux from the neural network onto the boundary using the specified method in subsection 4.3.1. Finally, once the timestep of 1.0 is reached, the if function in Line 21 results in true, whereas in lines 22-24, Modulus gets trained on the accumulated boundary conditions. After that, the collected boundary conditions are dropped since the solver will iterate multiple times over the timeframe by reading the checkpoint created in line 15 at the beginning of the simulation and specified by the preCICE configuration. The maximum amount of iterations is set to be 100. This continues until the model has iterated over the timeframe a hundred times while also training the NN that amount of times or the coupled FEniCS Neumann solver detects a convergence in a specified value range. Of note is also that the validators used in subsection 4.2.7 are identically implemented in the Helper class to provide data to compare both implementations.

5 Evaluation

This chapter will showcase and analyze the data generated during this thesis to assert the accuracy and efficiency of the uncoupled and coupled PINN implementations. First, the Validator plots generated with the code will be analyzed in section 5.1. Then, the training behavior will be examined in section 5.2.

5.1 Validating model output

If the models approximate the manufactured solution and the error margins they generally have, they will be shown using the form in the 4.2.6 implemented validator plot. Figure 5.1 shows the pure Modulus model plot.

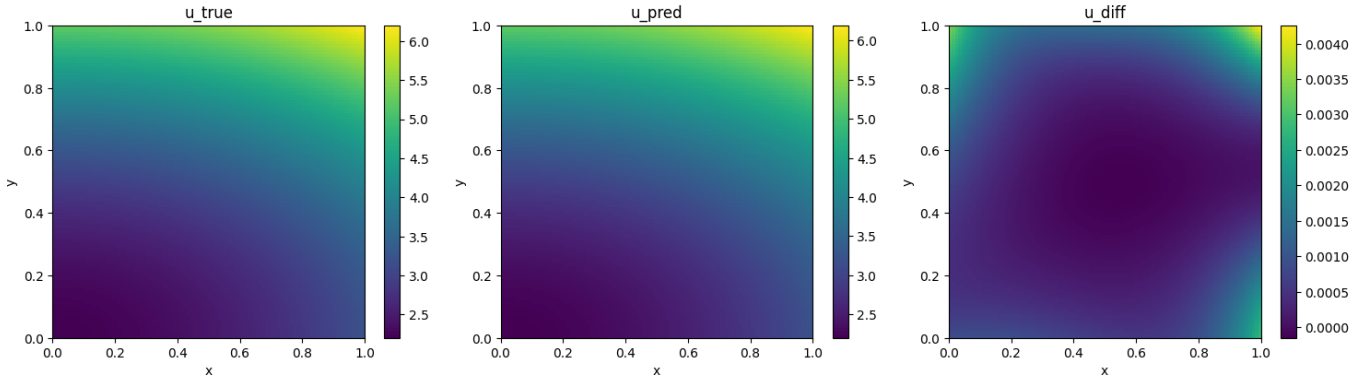


Figure 5.1 Trained models validator plot at t=1.0 after 100k steps training; manufactured solution(left), actual model prediction(middle), difference(right)

The model plot approximates the manufactured solution. It is moderately precise, with the most significant errors in the right plot being less than 0.005. However, compared to many numerical conventional solutions to this problem, it does compare worse since a solver using FEniCS can easily reach the limit of floating point precision. Now, Figure 5.2 shows the corresponding plot in the coupled setup.

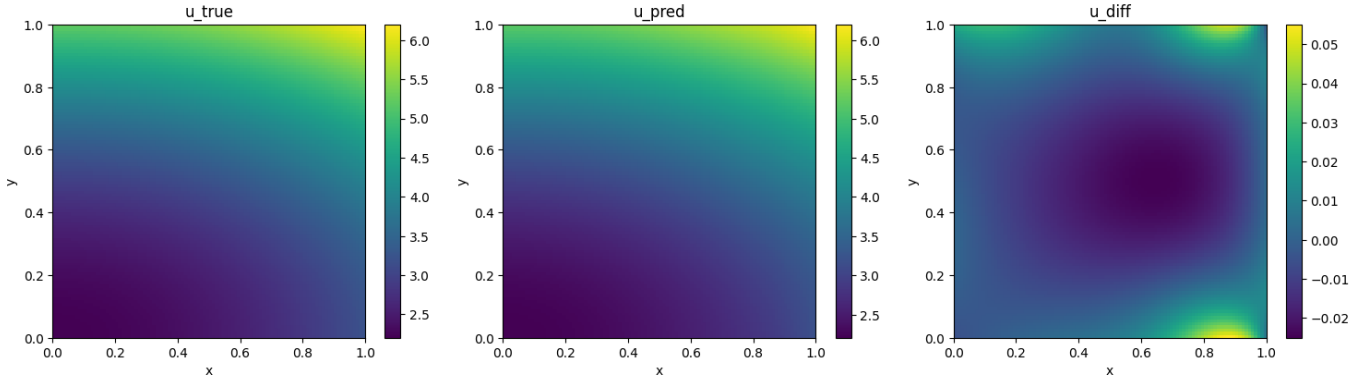


Figure 5.2 Trained models validator plot at t=1.0 after 100k steps training with Dirichlet Neuman coupling; manufactured solution(left), actual model prediction(middle), difference(right)

While the plot still follows the actual solution. The errors are more significant in this example, with the largest error being more than ten times that of the uncoupled implementation. This could result from the

training being unstable since the numerical solution, especially early on in the training, does not provide stable or accurate data. Alternatively, the current implementation is potentially not optimal for this setup since the most significant errors, according to the right plot, are found on the edges close to the coupling boundary but not touching it. Still, the most significant errors approach 0.06, and the model plot still approximates the manufactured solution.

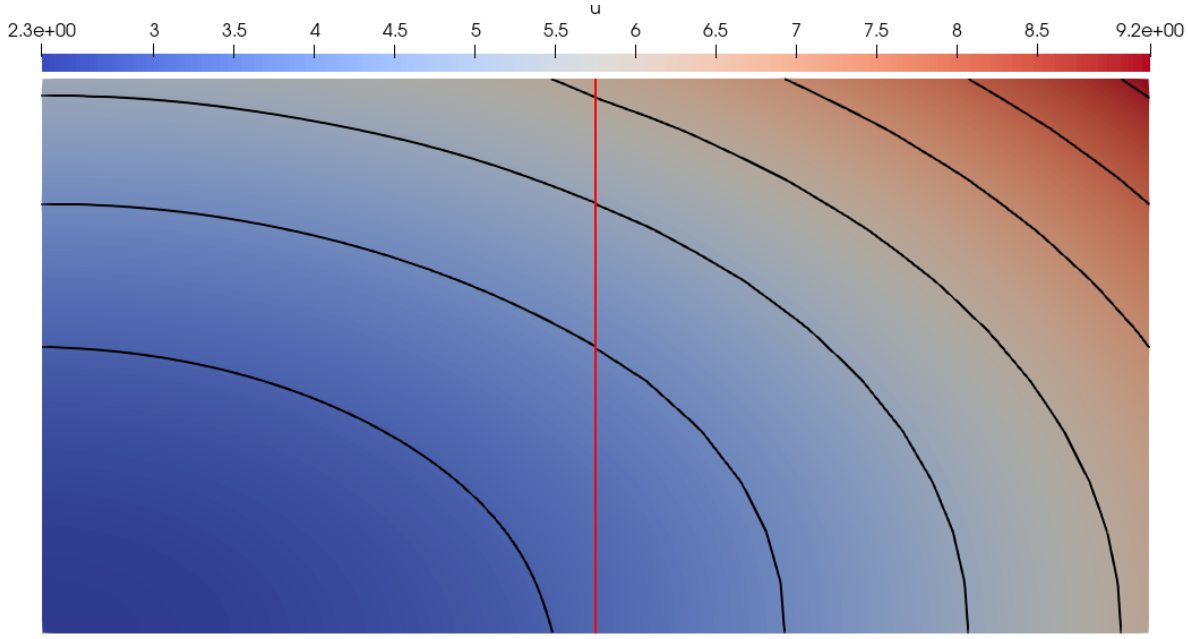


Figure 5.3 Resulting Dirichlet Modulus model (left of marked boundary) with Neumann FEniCS solver result (right of marked boundary). Showing the solution with contours at t=1.0 with marked boundary

The interesting part in this plot is if the contours generated are consistent across the coupling boundary; this can indicate whether the coupling is creating issues. In a perfect coupling setup, the contours will remain consistent across the boundary in this experiment. The main issue is that a Delaunay filter must be applied to generate Modulus data since the Point data from Modulus cannot be directly processed. This, however, can create inaccuracies since it only approximates a plot. However, the contours are nearly meeting. They have a slight jump on the coupling boundary, indicating an inaccurate coupled solution or that the filter is creating too much noise. Since the manufactured solution provided in the Modulus data also leads to the same jump in contours, it can be assumed that the error could stem from the filter.

5.2 Evaluating training

Now, the training behavior will be summarized. First, the data found in the model logs and visualized with Tensorboard will be discussed. Then also the training times are mentioned.

5.2.1 Training convergence

Figure 5.4 shows the aggregation of the in section 2.3 defined loss functions across the entire training. It indicates how well the model approximates the training data, with a sharp drop early and continuous reduction with increasing training amount. In addition, Figure 5.6 shows the gradients of the Loss function calculated for the adjustment in the system explained in section 2.3, with it dropping very quickly.

The problems then occur when the coupled scenario is observed. In Figure 5.5, we can see a similar aggregated loss as in Figure 5.4, but with key differences that in the beginning, the coupled scenario has a massive drop-off, since the training only happens on the initial condition and then the latter parts of the loss have far higher values, which means the model does not predict the training data just as well. Additionally,

the coupled scenario has increased jumps every 1k steps because the coupled boundary gets updated, leading to different training data and an entirely different FEniCS function approximated. Finally, we can see with Figure 5.7 that the gradients were so prominent at the beginning of the training that Modulus internally limited them to 0.5. After 40k steps of training, it did return to smaller values. However, it shows that the training process was much more intensive in the early stages because all values had been changed far more drastically than in the uncoupled scenario.

5.2.2 Training time

The following runtimes have been recorded using the Tensorboard logging running on the in subsection 4.1.1 mentioned Hardware. The training of the standard setup in its current configuration took 42m 14s. On the other hand, the training of the coupled setup with the identical step count took 6h 57m 55s. This shows a significant slowdown when using the coupled scenario. While the coupled scenario has slightly more training data from having bigger batches on the coupling boundary, to enforce the ever-changing function there, it is not in ratio to the runtime. The most probable cause is a significant overhead because of the implementation where 100 different boundary constraints across the time range are created for the coupled solution, in addition to the slow, non-parallelized CPU-bound data generation from the FEniCS function.

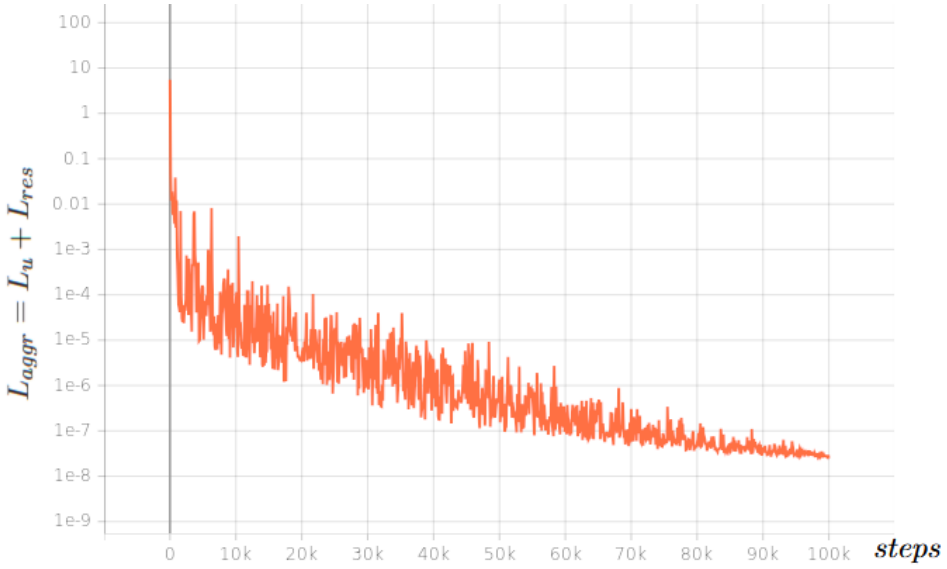


Figure 5.4 The uncoupled models aggregated loss function with logarithmic vertical axis

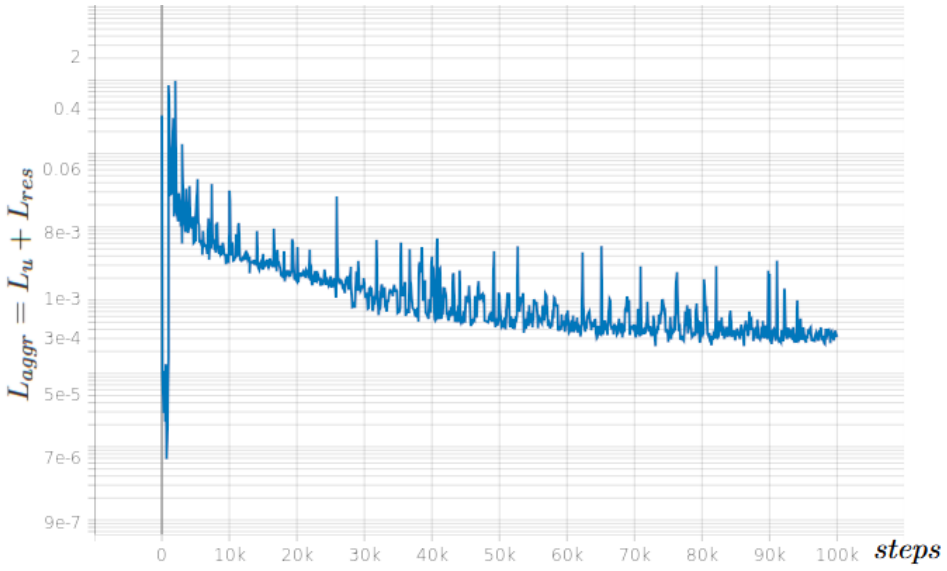


Figure 5.5 The coupled models aggregated loss function with logarithmic vertical axis

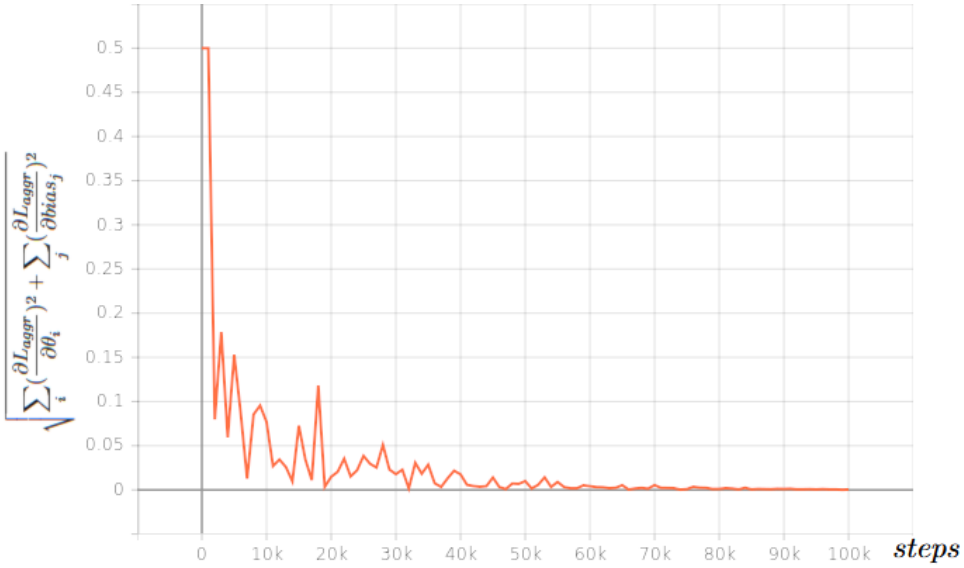


Figure 5.6 The uncoupled model's error grad norm, the average gradient of the error function on every model parameter

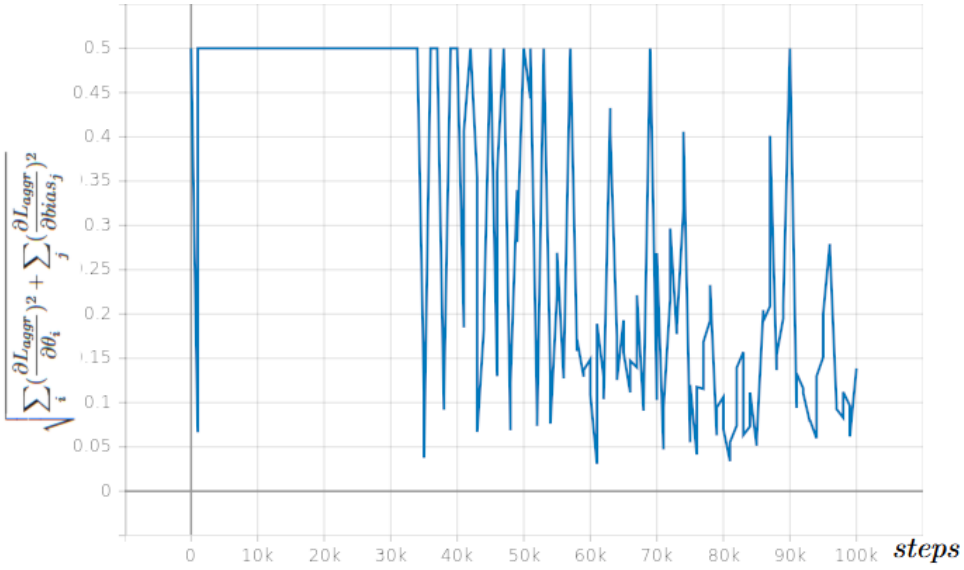


Figure 5.7 The coupled models error grad norm, the average gradient of the error function on every model parameter

6 Conclusion

In this final chapter, the findings and observations of this thesis concerning the main research question will be summarized and discussed in section 6.1. And then in section 6.2, an outlook is given on unresolved questions and potential future work.

6.1 Discussion

The in section 2.3 discussed PINN approach was implemented using the Nvidia Modulus library from section 3.1. In section 4.2, a model implementing the transient two-dimensional heat transfer was created. Which then was in the course of chapter 4 expanded upon to be able to couple through preCICE to a FEniCS solver for functioning Dirichlet-Neumann Coupling. With the analysis in chapter 5, both models approximate the manufactured solution reasonably. While not providing the same accuracy in the coupled scenarios as in the uncoupled setup, it is a working model trained by being coupled through preCICE. Whether it is feasible to couple a Modulus model through preCICE can be answered with a definitive yes. However, the accuracy of both models is far worse than any common comparable numerical solution given similar computing resources see subsection 5.2.2. In addition, the approach of training the model while coupling with another solver is potentially flawed, compared a solution where preliminary simulation data from conventional solutions is used to train a model to then couple without additional training to other solvers could yield far better results since the early numerical instability of most conventional solution would then not affect the training and would make a far more precise coupled NN as seen with the error margins in section 5.1.

6.2 Outlook

Now, gathering all the previously found insights, an outlook is given for potential future work on this topic.

One main interest is creating a dedicated Modulus adapter or a Modulus implementation using preCICE directly since the current FEniCS implementation slows down the training by requiring substantially longer for data generation. An analysis of how long that exactly is with an implementation that uses different approaches like parallelization can be used, or even a data-driven constraint being enforced on the coupled boundary constraint would be a potential improvement.

Another point is that the current implementation focuses on a relatively simple heat problem with a known manufactured solution. More diverse and complicated setups could unveil further flaws and limits of the Modulus software. Especially when coupling to different software, how the accuracy converges and performance behaves is also unanswered.

Finally, Modulus also includes many different models, hyperparameters, and settings inside its toolkit; research into them could provide more advanced and robust setups since this thesis only includes using an FCNN with the Tanh function. The possible combinations presented in Modulus are vast, and the optimal setups and settings for specific problems regarding efficiency and error margins are not quickly answered.

Bibliography

[1] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[2] C. Farhat, M. Lesoinne, and K. Pierson, "A scalable dual-primal domain decomposition method," *Numerical linear algebra with applications*, vol. 7, no. 7-8, pp. 687–714, 2000.

[3] G. Strang, *Wissenschaftliches Rechnen*. Springer-Verlag, 2010.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[5] H. P. Langtangen and A. Logg, *Solving PDEs in python: the FEniCS tutorial I*. Springer Nature, 2017.

[6] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in pytorch," 2017.

[7] 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*, vol. 378, pp. 686–707, 2019.

[8] B. Rodenberg, I. Desai, R. Hertrich, A. Jaust, and B. Uekermann, "FEniCS–preCICE: Coupling FEniCS to other simulation software," *SoftwareX*, vol. 16, p. 100 807, 2021, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2021.100807.

[9] B. Rüth, B. Uekermann, M. Mehl, P. Birken, A. Monge, and H.-J. Bungartz, "Quasi-newton waveform iteration for partitioned surface-coupled multiphysics applications," *International Journal for Numerical Methods in Engineering*, vol. 122, no. 19, pp. 5236–5257, 2021.

[10] G. Chourdakis, K. Davis, B. Rodenberg, *et al.*, "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]," *Open Research Europe*, vol. 2, no. 51, 2022. DOI: 10.12688/openreseurope.14445.2.

[11] F. M. Eler, A. A. de Souza, P. Couto, and A. L. Coutinho, "Evaluating different neural networks architectures for the solution of heat conduction problems in nvidia modulus," in *XLIII Ibero-Latin American Congress on Computational Methods in Engineering*, vol. 4, 2022.

[12] M. Zhou, G. Mei, and N. Xu, "Enhancing computational accuracy in surrogate modeling for elastic–plastic problems by coupling s-fem and physics-informed deep learning," *Mathematics*, vol. 11, no. 9, p. 2016, 2023.

[13] J. Chen, G. Chourdakis, I. Desai, *et al.*, *preCICE Distribution Version v2404.0*, version V1, 2024. DOI: 10.18419/DARUS-4167. [Online]. Available: https://doi.org/10.18419/DARUS-4167.

[14] H. Hu, L. Qi, and X. Chao, "Physics-informed neural networks (pinn) for computational solid mechanics: Numerical frameworks and applications," *Thin-Walled Structures*, p. 112 495, 2024.

Acronyms

FCNN Fully Connected Neural Network. 5, 16, 31

FEM Finite Element Method. 1, 3, 9, 11

FFNN Feedforward Neural Network. 5

NN Neural Networks. ix, 1, 5–7, 13, 15, 22, 23, 31

ODE Ordinary Differential Equations. ix, 1, 3, 16

PDE Partial Differential Equations. ix, 1, 3, 7, 11, 16–18

PINN Physics-Informed Neural Networks. ix, 1, 3, 5, 7, 9, 25, 31

Tanh Hyperbolic Tangent. 6, 16, 31, 37

List of Figures