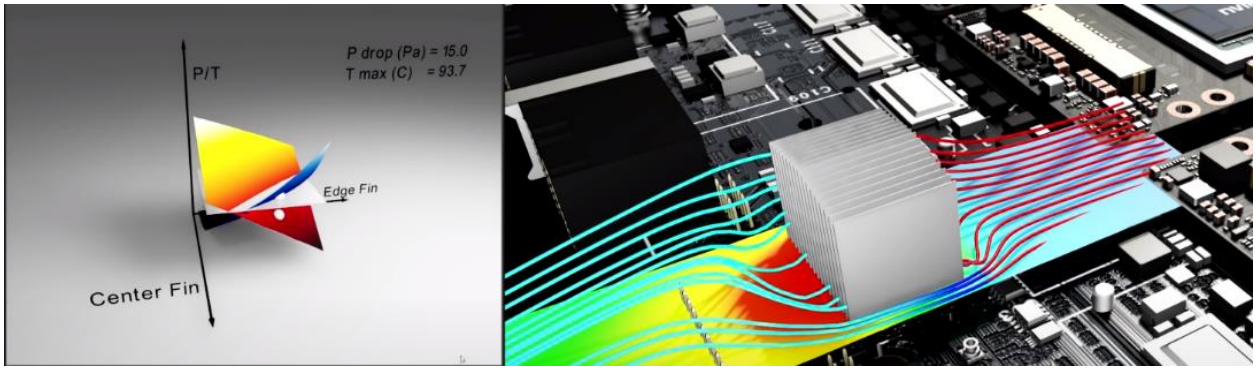
Modulus

A Neural Network Based Partial Differential Equation Solver



User Guide

Release v21.06 | November 9, 2021

NVIDIA

Notice

The information provided in this specification is believed to be accurate and reliable as of the date provided. However, NVIDIA Corporation ("NVIDIA") does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This publication supersedes and replaces all other specifications for the product that may have been previously supplied.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and other changes to this specification, at any time and/or to discontinue any product or service without notice. Customer should obtain the latest relevant specification before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer. NVIDIA hereby expressly objects to applying any customer general terms and conditions with regard to the purchase of the NVIDIA product referenced in this specification.

NVIDIA products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on these specifications will be suitable for any specified use without further testing or modification. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to ensure the product is suitable and fit for the application planned by customer and to do the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this specification. NVIDIA does not accept any liability related to any default, damage, costs or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this specification, or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this specification. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA. Reproduction of information in this specification is permissible only if reproduction is approved by NVIDIA in writing, is reproduced without alteration, and is accompanied by all associated conditions, limitations, and notices.

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the NVIDIA terms and conditions of sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, CUDA, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

Contents

MODULUS USER GUIDE

A NEURAL NETWORK BASED PARTIAL DIFFERENTIAL EQUATIONS SOLVER

Getting Started: Guidelines on using this document and System Requirements

Structure of the User Guide

This User Guide gives you a headstart in solving your own physics-based problems using neural networks. Chapter 1 provides a brief description of the theory of PINNs (Physics Informed Neural Networks) used in Modulus. It outlines the PINNs approach of solving PDEs (Partial Differential Equations) and feature improvements in Modulus over the standard PINNs.

Tutorials 2 and 3, on Lid Driven Cavity flow, are intended to give you a thorough description of the Modulus user interface. Having solved this canonical problem, you can easily draw parallels between other PDE solvers and Modulus which will aid you in setting up your own problems. Tutorial 4 solves a transient 1D wave equation and demonstrates coding a custom PDE in Modulus. The time-dependent problem is solved using the continuous-time, time marching schedules, RNN (Recurrent Neural Networks) and GRU (Gated Response Units) approaches. Tutorial 5 applies the concepts of continuous time for a 2d wave propagation problem encountered in seismic surveys. Tutorial 6 introduces Modulus' sequential solver and solves the canonical Taylor-Green vortex decay problem using the moving time window approach. Tutorial 7 shows the use of Modulus for solving a system of ordinary differential equations. Tutorial 8 simulates an advection-diffusion problem to model a scalar transport phenomenon. In tutorial 9 we show how to solve the PDEs in their variational form (weak solutions) using Modulus. Such formulation helps to solve the PDEs for which obtaining the solution in classical sense is very complex (e.g. problems with interface, singularities, etc.).Tutorials 10 and 11 introduce a new physics: Tutorial 10 covers the electromagnetic simulations using PINNs, solving the frequency domain Maxwell's equations while tutorial 11 uses PINNs to solve various 3D and 2D stress-strain problems.

Tutorial 12 illustrates tuning the neural network hyperparameters and also marks the transition to more complex problems. Tutorial 13 solves a multi-physics problem involving a conjugate heat transfer with interface boundary conditions on a 3D 3-fin geometry. Real world geometrical shapes are often designed using a CAD program and are exported using one of the tessellated (e.g. STL, OBJ), neutral (IGES, STEP) or native CAD formats. Tutorial 14 demonstrates import of an STL geometry (that can be exported from a CAD program) in Modulus. In this tutorial, Modulus uses its native SDF (Signed Distance Function) library to calculate the SDF for the points in the point cloud and determine if they are on, outside or inside the surface. Tutorial 15 guides you in assimilating data from a point cloud using the CSV files. Additionally, tutorial 15 also provides a guide on using PINNs to assimilate the known quantities to infer or invert data which would be otherwise impossible for traditional methods.

In tutorials 16 and beyond, we address some of the more advanced 3D applications of the PINNs like solving parameterized geometry, design optimization, and multi-GPU performance. Tutorial 16 demonstrates a design optimization problem where a six variable, parameterized flow is solved for a 3-fin heat sink to determine the lowest temperature for a given pressure drop. Tutorial 16 showcases the major computational advantage of PINNs in solving industry-scale design optimization problems. Next, there is a case study on an FPGA heat sink (tutorial 17) that showcase the various features and architectures in Modulus for more complex geometry. Tutorial 18 shows an even more complicated geometry with real physics. Such problems present a new class of complexities for the PINNs and we discuss the use of algorithms like hFTB (heat transfer coefficient forward temperature backward), gradient aggregation and surrogate modeling through gPC (generalized polynomial chaos) to tackle them. Finally, performance for single GPU and scalability for multi-GPU/node cases is covered in tutorial 19.

With a broad array of examples covered, we intend to give you, a glimpse into the possibilities with Modulus. Each example comes with validation data either from analytical solutions or open-source solvers (e.g. OpenFOAM for CFD problems) for you to compare the PINNs results. You can choose an example from the user guide that resembles your problem, and modify it to solve it using Modulus library.

What's new in Modulus 21.06

Modulus v21.06 consists of several enhancements in terms of new features, physics and solution methodologies. A summary of the additions and changes compared to v20.12 is given below:

1. Chapter 1: Description of theoretical underpinnings of new features like Gradient Aggregation, Multiplicative Filter Network, Homoscedastic task uncertainty for loss weighting, Generalized Polynomial Chaos, etc.
2. Tutorial 4: Additional schemes on solving transient problems through temporal loss weighting and time-marching schedule.
3. Tutorial 5: Solution to a 2D seismic wave propagation commonly used in seismic survey.
4. Tutorial 6: Solution to the transient Taylor Green vortex decay problem using the moving time window approach.
5. Tutorial 10: Electromagnetic simulations with features to solve the Frequency domain Maxwell's equations in scalar and vector form.
6. Tutorial 14: Details on how to leverage transfer learning for faster training of new patient-specific models.
7. Tutorial 18: Application of Modulus for an industrial problem with real properties and physics. Solving the conjugate heat transfer problem using hFTB algorithm and application of Generalized Polynomial Chaos for design optimization.

Targeted Users

There are two ways of using this guide:

1. If you wish to dive deep in the field of Physics Informed Neural Networks (PINNs) and understand the key concepts, we recommend you to start with Chapter 1. This chapter covers the theory behind the PINNs as well as some of the feature improvements we have made over the standard PINNs to improve the robustness and speed. Next, you can step through a quick tutorial on Lid Driven Cavity flow (tutorial 2 and 3) to get familiarized with the Modulus user interface and the various modules available to define geometry, equations, and boundary conditions. From then on, you can refer to the categorically sorted tutorials based on the Physics and problems you wish to simulate.
2. If you want to use Modulus to solve PDEs without exploring the depths of PINNs, we recommend you to skip tutorial 1 and start directly with Lid Driven Cavity flow (tutorials 2 and 3) to get acquainted with the process of setting up a problem in Modulus. For information on steps to define a custom PDE, one can refer to tutorial 4.

We strongly recommend you to visit tutorial 2 before using Modulus for custom problems.

Note: If you have difficulties copying code from the snippets presented in these tutorials, you can refer to the scripts in the `examples` directory and use the tutorials as a guide for those scripts. All the scripts used for the examples in the tutorials as well as other additional examples can be found in the `examples/` directory of the Modulus repository (a complete list can be found in section 1.4 of tutorial 1).

System Requirements

Table 1: System Configuration

Operating System	• Ubuntu 18.04 or Linux 4.18 kernel
Driver and GPU Requirements	• • Bare Metal version: NVIDIA driver 465.19 required only if SDF library is used • • Docker container: NVIDIA driver 465.19 or higher driver must be used. If using a Tesla (for example, T4 or any other Tesla board), you may use NVIDIA driver release 440.30 or 418.xx however any drivers older than 465 will not support the SDF library. (https://docs.nvidia.com/deeplearning/frameworks/support-matrix/index.html)
Required installations for Bare Metal version	• Python 3.6 • Tensorflow 1.15 • Horovod 0.21.0
Supported Processors	• 64-bit x86 (this dependency is only when the SDF library is used since the SDF library is compiled on x86. If you need the SDF compiled on Power9 architecture then please e-mail us at: modulus-team@exchange.nvidia.com) • NVIDIA GPU based on the following architectures: – Nvidia Ampere GPU Architecture (A100) – Volta (V100, Titan V, Quadro GV100) – Turing (T4, Quadro RTX series) – Pascal (P100, P40, P4, Titan Xp, Titan X) **All studies in the User Guide are done using V100 on DGX-1. A100 has also been tested.**

NOTE: To get the benefits of all the performance improvements (e.g. AMP, multi-GPU scaling, etc.), use the NVIDIA Tensorflow container for Modulus. This container comes with all the prerequisites and dependencies and allows you to get started efficiently with Modulus.

1 Theory

1.1 Introduction

In this tutorial/guide, we will walk through the following topics:

1. Theory of neural network differential equation solvers
2. Important modifications to the standard neural network solvers
3. Overview of Code and Examples in Modulus

1.2 Neural Network Solver Methodology

In this section we provide a brief introduction to solving differential equations with neural networks. The idea is to use a neural network to approximate the solution to the given differential equation and boundary conditions. We train this neural network by constructing a loss function for how well the neural network is satisfying the differential equation and boundary conditions. If the network is able to minimize this loss function then it will in effect, solve the given differential equation.

To illustrate this idea we will give an example of solving the following problem,

$$\mathbf{P}: \begin{cases} \frac{\delta^2 u}{\delta x^2}(x) = f(x), \\ u(0) = u(1) = 0, \end{cases} \tag{1}$$

We start by constructing a neural network $u_{net}(x)$. The input to this network is a single value $x \in \mathbb{R}$ and its output is also a single value $u_{net}(x) \in \mathbb{R}$. We suppose that this neural network is infinitely differentiable, $u_{net} \in C^{\infty}$. The typical neural network used is a deep fully connected network where the activation functions are infinitely differentiable.

Next we need to construct a loss function to train this neural network. We easily encode the boundary conditions as a loss in the following way:

$$L_{BC} = u_{net}(0)^2 + u_{net}(1)^2 \tag{2}$$

For encoding the equations, we need to compute the derivatives of u_{net}. Using automatic differentiation we can do so and compute $\frac{\delta^2 u_{net}}{\delta x^2}(x)$. This allows us to write a loss function of the form:

$$L_{residual} = \frac{1}{N} \sum_{i=0}^{N} \left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i) \right)^2 \tag{3}$$

Where the x_i's are a batch of points sampled in the interior, $x_i \in (0, 1)$. Our total loss is then $L = L_{BC} + L_{residual}$. Optimizers such as Adam [1] are used to train this neural network. Given $f(x) = 1$, the true solution is $\frac{1}{2}(x - 1)x$. Upon solving the problem, you can obtain good agreement between the exact solution and the neural network solution as shown in Figure 1.

1.3 Physics Informed Neural Networks in Modulus

Modulus is a neural network solver that can solve complex problems with intricate geometries and multiple physics. In order to achieve this we have deviated and improved on the current state-of-the-art in several important ways. In this section we will briefly cover some topics related to this.

1.3.1 Integral Formulation of losses

In literature, the losses are often defined as a summation similar to our above equation 3, [2]. In Modulus, we take a different approach and view the losses as integrals. You can instead write $L_{residual}$ in the form,

$$L_{residual} = \int_0^1 \left(\frac{\delta^2 u_{net}}{\delta x^2}(x) - f(x) \right)^2 dx \tag{4}$$

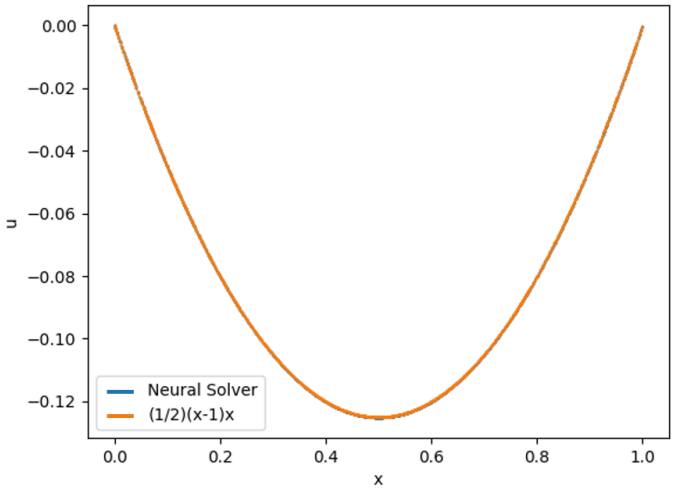


Figure 1: Neural Network Solver compared with analytical solution.

Now there is a question of how we approximate this integral. We can easily see that if we use Monte Carlo integration we arrive at the same summation in equation 3.

$$\int_0^1 \left(\frac{\delta^2 u_{net}}{\delta x^2}(x) - f(x)\right)^2 dx \approx (\int_0^1 dx)\frac{1}{N}\sum_{i=0}^{N}\left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i)\right)^2 = \frac{1}{N}\sum_{i=0}^{N}\left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i)\right)^2 \quad (5)$$

We note that, this arrives at the exact same summation because $\int_0^1 dx = 1$. However, this will scale the loss proportional to the area. We view this as a benefit because it keeps the loss per area consistent across domains. We also note that this opens the door to more efficient integration techniques. In several examples, in this user guide we sample with higher frequency in certain areas of the domain to approximate the integral losses more efficiently.

1.3.2 Integral Equations

Many PDEs of interest have integral formulations. Take for example the continuity equation for incompressible flow,

$$\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} + \frac{\delta w}{\delta z} = 0 \quad (6)$$

We can write this in integral form as the following,

$$\oiint_S (n_x u + n_y v + n_z w)\,\mathrm{d}S = 0 \quad (7)$$

Where S is any closed surface in the domain and n_x, n_y, n_z are the normals. We can construct a loss function using this integral form and approximate it with Monte Carlo Integration in the following way,

$$L_{IC} = \left(\oiint_S (n_x u + n_y v + n_z w)\,\mathrm{d}S\right)^2 \approx \left((\oiint_S \mathrm{d}S)\frac{1}{N}\sum_{i=0}^{N}(n_x^i u_i + n_y^i v_i + n_z^i w_i)\right)^2 \quad (8)$$

For some problems we have found that integrating such losses significantly speeds up convergence.

1.3.3 Parameterized Geometries

One important advantage of a neural network solver over traditional numerical methods is its ability to solve parameterized geometries [3]. To illustrate this concept we solve a parameterized version of equation 1. Suppose we want to know how the solution to this equation changes as we move the position on the boundary condition $u(l) = 0$. We can parameterize this position with a variable $l \in [1, 2]$ and our equation now has the form,

$$\mathbf{P}: \begin{cases} \frac{\delta^2 u}{\delta x^2}(x) = f(x), \\ u(0) = u(l) = 0, \end{cases} \tag{9}$$

To solve this parameterized problem we can have the neural network take l as input, $u_{net}(x, l)$. The losses then take the form,

$$L_{residual} = \int_1^2 \int_0^l \left(\frac{\delta^2 u_{net}}{\delta x^2}(x, l) - f(x) \right)^2 dxdl \approx \left(\int_1^2 \int_0^l dxdl \right) \frac{1}{N} \sum_{i=0}^{N} \left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i, l_i) - f(x_i) \right)^2 \tag{10}$$

$$L_{BC} = \int_1^2 (u_{net}(0, l))^2 + (u_{net}(l, l)dl \approx \left(\int_1^2 dl \right) \frac{1}{N} \sum_{i=0}^{N} (u_{net}(0, l_i))^2 + (u_{net}(l_i, l_i))^2 \tag{11}$$

In figure 2 we see the solution to the differential equation for various l values after optimizing the network on this loss. While this example problem is overly simplistic, the ability to solve parameterized geometries presents significant industrial value. Instead of performing a single simulation we can solve multiple designs at the same time and for reduced computational cost. Examples of this will be given later in the user guide.

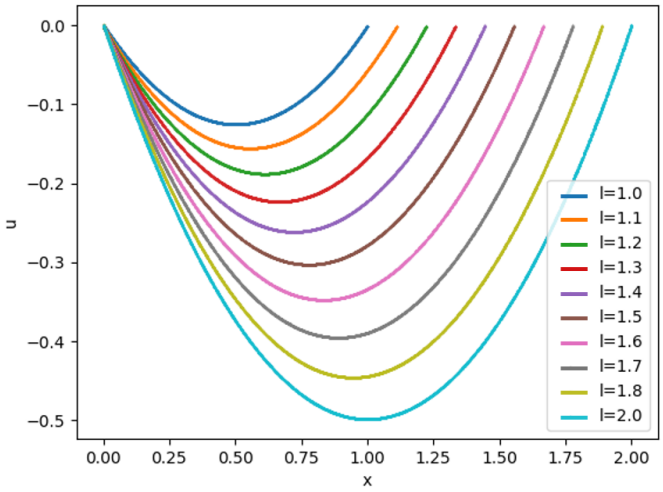


Figure 2: Modulus solving parameterized differential equation problem.

1.3.4 Inverse Problems

Another useful application of a neural network solver is solving inverse problems. In an inverse problem, we start with a set of observations and then use those observations to calculate the causal factors that produced them. To illustrate how to solve inverse problems with a neural network solver, we give the example of inverting out the source term $f(x)$ from equation 1. Suppose we are given the solution $u_{true}(x)$ at 100 random points between 0 and 1 and we want to determine the $f(x)$ that is causing it. We can do this by making two neural networks $u_{net}(x)$ and $f_{net}(x)$ to approximate both $u(x)$ and $f(x)$. These networks are then optimized to minimize the following losses;

$$L_{residual} \approx \left(\int_0^1 dx \right) \frac{1}{N} \sum_{i=0}^{N} \left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i, l_i) - f_{net}(x_i) \right)^2 \tag{12}$$

$$L_{data} = \frac{1}{100} \sum_{i=0}^{100} (u_{net}(x_i) - u_{true}(x_i))^2 \tag{13}$$

Using the function $u_{true}(x) = \frac{1}{48}(8x(-1+x^2) - (3sin(4\pi x))/\pi^2)$ the solution for $f(x)$ is $x + sin(4\pi x)$. We solve this problem and compare the results in figure 3, 4

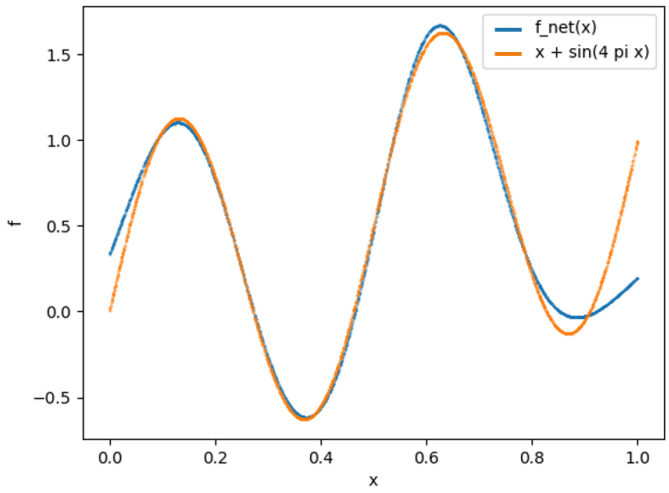


Figure 3: Comparison of true solution for $f(x)$ and the function approximated by the NN.

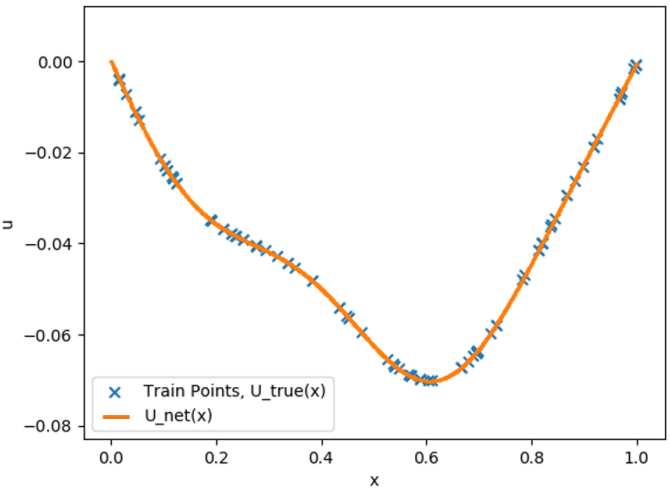


Figure 4: Comparison of $u_{net}(x)$ and train points from u_{true}.

1.4 The examples/ directory

With the Modulus package, we include a wide array of pre-tested examples, and implementations of the modulus library applied to a varied class of problems. Some of these example files are later used in this manual for the more detailed and guided explanation.

The current package of Modulus will include the following example files:

Directory	File/Scripts	Description
ldc/	ldc_2d.py	2D Laminar Flow, Re 10
	ldc_2d_zeroEq.py	2D Turbulent Flow, Zero Equation Turbulence Model, Re 1000
annular_ring/	annular_ring.py	2D Laminar Flow, Re 100
	annular_ring_parameterized.py	2D Laminar Flow, Parametric Geometry, Re 100
chip_2d/	chip_2d.py	2D Laminar Flow, Re 50
cylinder_2d/	cylinder_2d.py	2D Laminar Flow, Exact Mass Balance (Continuity) Constraint, Re 50
wave_equation/	wave_1d.py	1D Transient Wave Equation
	wave_1d_rnn.py wave_1d_gru.py	Solution to the 1D Transient Wave Equation using Experimental RNN and GRU Networks
	wave_1d_inverse.py	1D Inverse Problem, (finding wave speed)
ode_spring_mass/	spring_mass_solver.py	Coupled Spring Mass system, Ordinary Differential Equations
discontinuous_galerkin/	Multiple Scripts	Variational Method (Weak Solutions using PINNs) for problems with interface, point source, etc.
plane_displacement/	plane_displacement.py	2D Plane stress problem in Variational form, Linear Elasticity
fuselage_panel/	Multiple Scripts	2D Plane Stress, Parameterized Hoop Stress, Linear Elasticity
bracket/	bracket.py	3D Linear Elasticity
aneurysm/	aneurysm.py	3D Laminar Flow inside a complex Geometry (STL import)
three_fin_2d/	heat_sink.py	2D Heat Transfer, Re 100, Pr 5
	heat_sink_inverse.py	2D Inverse Problem, (finding diffusivity and viscosity)
three_fin_3d/	Multiple Scripts	Laminar, Turbulent and Parameterized implementations of 3D Conjugate Heat Transfer on a 3-Fin Heat Sink with Design Optimization, Re 50 and 500
fpga/	Multiple Scripts	Laminar, Turbulent and Parameterized implementations of 3D Conjugate Heat Transfer on a 17-Thin-Fins Heat Sink, Re 50 and 13,239.6. Implementation of Modulus' several Advanced Features
surface_pde/	surface.py torus.py	Examples of solving PDEs on 3D surfaces
seismic_wave/	wave_2d.py	2D Transient solution to a seismic wave propagation problem
taylor_green/	taylor_green.py	3D Transient solution to Taylor Green vortex decay problem using moving window approach
waveguide/	Multiple Scripts	2D and 3D Electromagnetics simulations. Maxwell's equations in Frequency domain
limerock/	Multiple Scripts	NVSwitch Heatsink simulation with real physics. Application of hFTB algoritm and surrogate modeling/design optimization through input parameterization and Generalized Polynomial Chaos.

Note: The details of the files in the modulus library can be found in the source code documentation.

1.5 Tips on improving accuracy and convergence speed of PINNs results

In this section, we will see the benefits of some of the key improvements we suggested in Section 1.3. Some of the improvements like adding integral continuity planes, weighting the losses spatially, and varying the point density in the areas of interest, have been key in making Modulus robust and capable of handling some of the larger-scale problems in reasonable time-frame.

1.5.1 Integral Continuity Planes

For some of the fluid flow problems involving channel flow, we found that, in addition to solving the Navier-Stokes equations in differential form, specifying the mass flow through some of the planes in the domain significantly speeds up the rate of convergence and gives better accuracy. Assuming there is no leakage of flow, we can guarantee that the flow exiting the system must be equal to the flow entering the system. Also, we found that by specifying such constraints at several other planes in the interior improves the accuracy further. For incompressible flows, one can replace mass flow with the volumetric flow rate.

Figure 5 shows the comparison of adding more integral continuity planes and points in the interior, applied to a problem of solving flow over a 3D 3-fin heat sink in a channel (tutorial 16). The one IC plane case has just one IC plane at the outlet while the 11 IC plane case has 10 IC planes in the interior in addition to the IC plane at the outlet. A lower mass imbalance inside the system indicates that the case run with 11 integral continuity planes helps in satisfying the continuity equation better and faster.

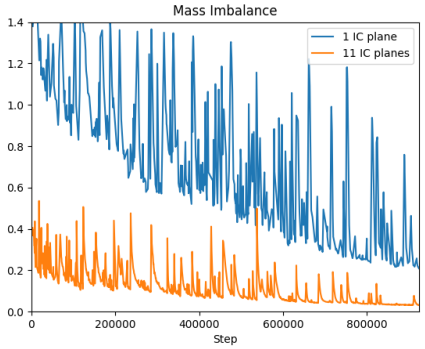


Figure 5: Improvements in accuracy by adding more Integral continuity planes and points inside the domain

1.5.2 Spatial Weighting of Losses (SDF weighting)

One area of considerable interest is weighting the losses with respect to each other. For example, we can weight the losses from equation 1 in the following way,

$$L = \lambda_{BC} L_{BC} + \lambda_{residual} L_{residual} \tag{14}$$

Depending on the λ_{BC} and $\lambda_{residual}$ this can impact the convergence of the solver. We can extend this idea to varying the weightings spatially as well. Written out in the integral formulation of the losses we get,

$$L_{residual} = \int_0^1 \lambda_{residual}(x) \left(\frac{\delta^2 u_{net}}{\delta x^2}(x) - f(x) \right)^2 dx \tag{15}$$

The choice for the $\lambda_{residual}(x)$, can be varied based on problem definition, and is an active field of research. In general, we have found it beneficial to weight losses lower on sharp gradients or discontinuous areas of the domain. For example, if there are discontinuities in the boundary conditions we may have the loss decay to 0 on these discontinuities. Another example is weighting the equation residuals by the signed distance function, SDF, of the geometries. If the geometry has sharp corners this often results in sharp gradients in the solution of the differential equation. Weighting by the SDF tends to weight these sharp gradients lower and often results in a convergence speed increase and sometimes also improved accuracy. In this user guide there are many examples of this and we defer further discussion to the specific examples.

Figure 6 shows L_2 errors for one such example of laminar flow (Reynolds number 50) over a 17-fin heat sink (tutorial 17) in the initial 100,000 iterations. The multiple closely spaced thin fins lead to several sharp gradients in flow equation residuals in the vicinity of the heat sink. Weighting them spatially, we essentially minimize the dominance of these sharp gradients during the iterations and achieve a faster rate of convergence.

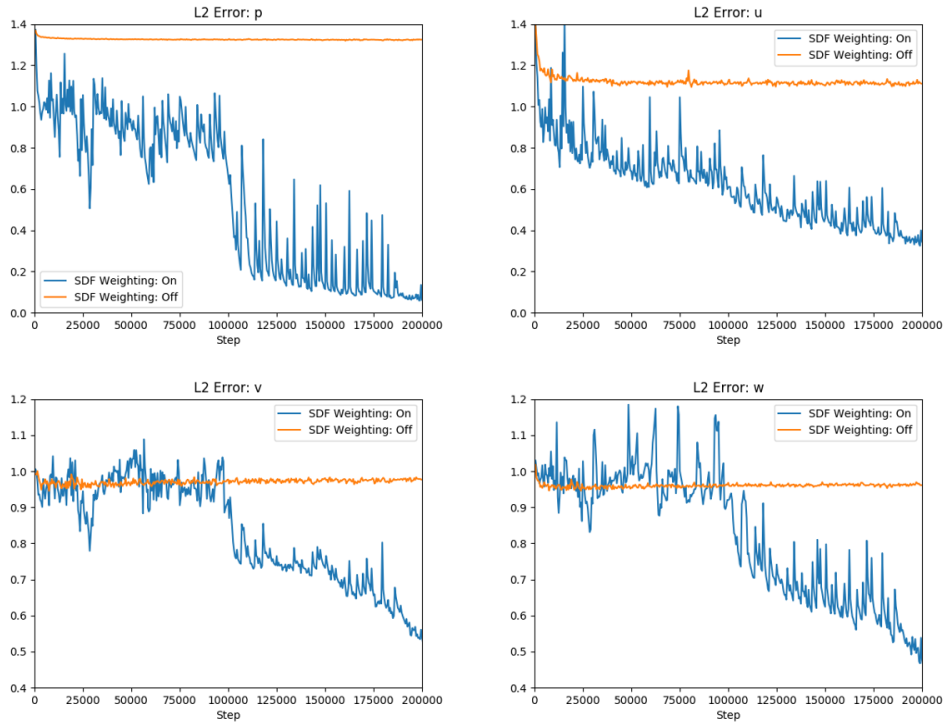


Figure 6: Improvements in convergence speed by weighting the equation residuals spatially.

A similar weighting is also applied to the intersection of boundaries. We will cover this in detail in the first tutorial on the Lid Driven Cavity flow (tutorial 2).

1.5.3 Increasing the Point cloud density

In this section, we discuss the accuracy improvements by adding more points in the areas where the field is expected to show a stronger spatial variation. This is somewhat similar to the FEM/FVM approach where the mesh density is increased in the areas where we wish to resolve the field better. If too few points are used when training then an issue can occur where the network may be satisfying the equation and boundary conditions correctly on these points but not in the spaces between these points. Quantifying the required density of points needed is an open research question however in practice if the validation losses or the validation residuals losses start to increase towards the end of training then more points may be necessary.

Figure 7 shows the comparison of increasing the point density in the vicinity of the same 17-fin heat sink (Figure 76) that we saw in the earlier comparison in Section 1.5.2, but now with a Reynolds number of 500 and with zero equation turbulence. Using more points near the heat sink, we are able to achieve better L_2 errors for p, v, and w.

Note: Care should be taken while increasing the integral continuity planes and adding more points in the domain as one might run into memory issues while training. If one runs into such an issue, some ways to avoid that would be to reduce the points sampled during each batch and increasing the number of GPUs. With the `horovod` implementation, the total $batchsize$ we define during the problem definition is the $batchsize/GPU$. A way to increase the total batch size would be to use more number of GPUs. Another way is to use gradient aggregation, which is discussed next.

1.5.4 Gradient Aggregation

As mentioned in the previous subsection, training of a neural network solver for complex problems requires a large batch size that can be beyond the available GPU memory limits. Increasing the number of GPUs can effectively increase

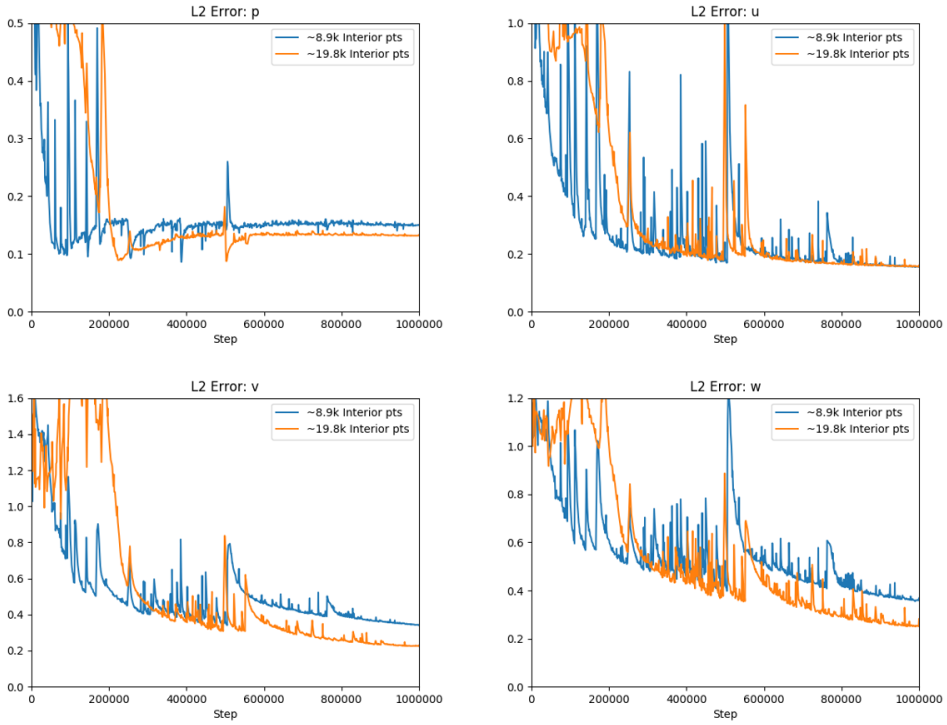


Figure 7: Improvements in accuracy by adding more points in the interior.

the batch size, however, one can instead use gradient aggregation in case of limited GPU availability. With gradient aggregation, the required gradients are computed in several forward/backward iterations using different mini batches of the point cloud and are then aggregated and applied to update the model parameters. This will, in effect, increase the batch size, although at the cost of increasing the training time. In the case of multi-GPU/node training, gradients corresponding to each mini-batch are aggregated locally on each GPU, and are then aggregated globally just before the model parameters are updated. Therefore, gradient aggregation does not introduce any extra communication overhead between the workers. Details on how to use the gradient aggregation in Modulus is provided in Tutorial 12.

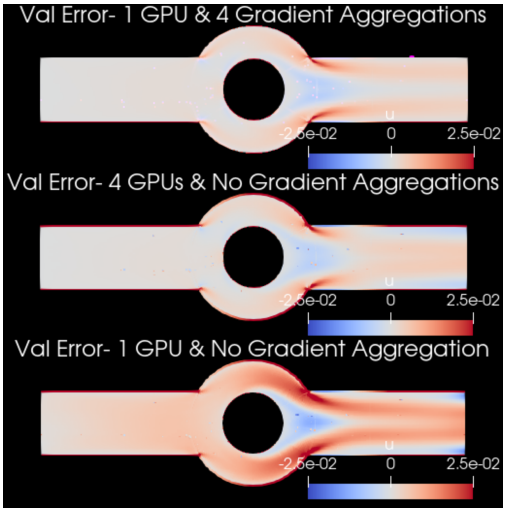


Figure 8: Increasing the batch size can improve the accuracy of neural network solvers. Results are for the u-velocity of an annular ring example trained with different number of GPUs and gradient aggregations.

1.5.5 Exact Continuity

Velocity-pressure formulations are the most widely used formulations of the Navier-Stokes equation. However, this formulation has two issues that can be challenging to deal with. The first is the pressure boundary conditions, which are not given naturally. The second is the absence of pressure in the continuity equation, in addition to the fact that there is no evolution equation for pressure that may allow to adjust mass conservation. A way to ensure mass conservation is the definition of the velocity field from a vector potential:

$$\vec{V} = \nabla \times \vec{\psi} = \left(\frac{\partial \psi_z}{\partial y} - \frac{\partial \psi_y}{\partial z}, \frac{\partial \psi_x}{\partial z} - \frac{\partial \psi_z}{\partial x}, \frac{\partial \psi_y}{\partial x} - \frac{\partial \psi_x}{\partial y} \right)^T, \tag{16}$$

where $\vec{\psi} = (\psi_x, \psi_y, \psi_z)$. This definition of the velocity field ensures that it is divergence free and that it satisfies continuity:

$$\nabla \cdot \vec{V} = \nabla \cdot (\nabla \times \vec{\psi}) = 0. \tag{17}$$

A good overview of related formulations and their advantages can be found in [4].

1.5.6 Importance Sampling

Suppose our problem is to find the optimal parameters $\boldsymbol{\theta}^*$ such that the Monte Carlo approximation of the integral loss is minimized

$$\begin{aligned} \boldsymbol{\theta}^* &= \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, \mathbb{E}_f \left[\ell(\boldsymbol{\theta}) \right] \\ &\approx \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, \frac{1}{N} \sum_{i=1}^{N} \ell(\boldsymbol{\theta}; \boldsymbol{x}_i), \quad \boldsymbol{x}_i \sim f(\boldsymbol{x}), \end{aligned} \tag{18}$$

where f is a uniform probability density function. In importance sampling, the sampling points are drawn from an alternative sampling distribution q such that the estimation variance of the integral loss is reduced, that is

$$\boldsymbol{\theta}^* \approx \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \, \frac{1}{N} \sum_{i=1}^{N} \frac{f(\boldsymbol{x}_i)}{q(\boldsymbol{x}_i)} \ell(\boldsymbol{\theta}; \boldsymbol{x}_i), \quad \boldsymbol{x}_i \sim q(\boldsymbol{x}). \tag{19}$$

Modulus offers point cloud importance sampling for improved convergence and accuracy, as originally proposed in [5]. In this scheme, the training points are updated adaptively based on a sampling measure q for a more accurate unbiased approximation of the loss, compared to uniform sampling. Details on the importance sampling implementation in Modulus are presented in Tutorial 12. Figure 9 shows a comparison between the uniform and importance sampling validation error results for the annular ring example, showing better accuracy when importance sampling is used. Here in this example, the training points are sampled according to a distribution proportional to the 2-norm of the velocity derivatives. The sampling probability computed at iteration 100K is also shown in Figure 10.

1.5.7 Quasi-Random Sampling

Training points in Modulus are generated according to a uniform distribution by default. An alternative to uniform sampling is the quasi-random sampling, which provides the means to generate training points with a low level of discrepancy across the domain. Among the popular low discrepancy sequences are the Halton sequences [6], the Sobol sequences, and the Hammersley sets, out of which the Halton sequences are adopted in Modulus. A snapshot of a batch of training points generated using uniform sampling and Halton sequences for the annular ring example is shown in Figure 11. Details on how to enable Halton sequences for sample generation are presented in Section 12.8. A case study on the use of Halton sequences to solve a conjugate heat transfer example is also presented in tutorial 17.

1.5.8 Symmetry Boundaries

In training of PINNs for problems with symmetry in geometry and physical quantities, reducing the computational domain and using the symmetry boundaries can help with accelerating the training, reducing the memory usage, and in some cases, improving the accuracy. In Modulus, the following symmetry boundary conditions at the line or plane of symmetry may be used:

- Zero value for the physical variables with odd symmetry.

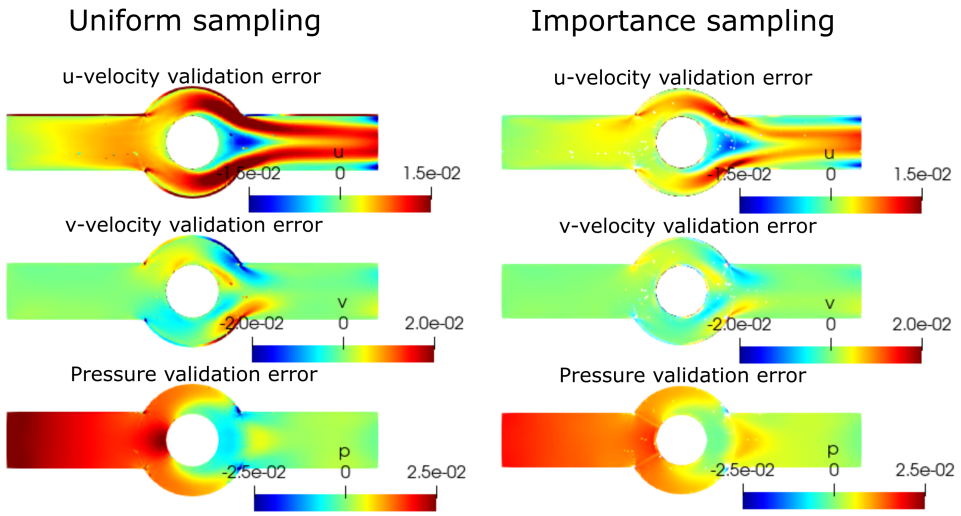


Figure 9: A comparison between the uniform and importance sampling validation error results for the annular ring example.

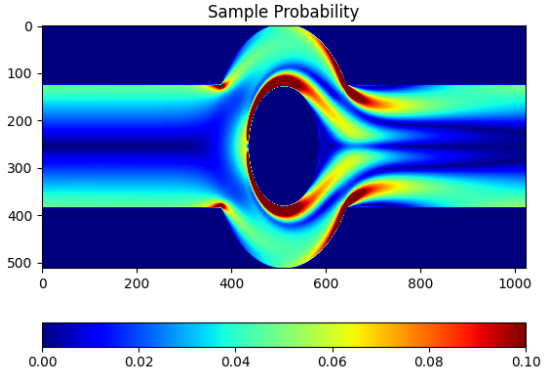


Figure 10: A visualization of the training point sampling probability at iteration 100K for the annular ring example.

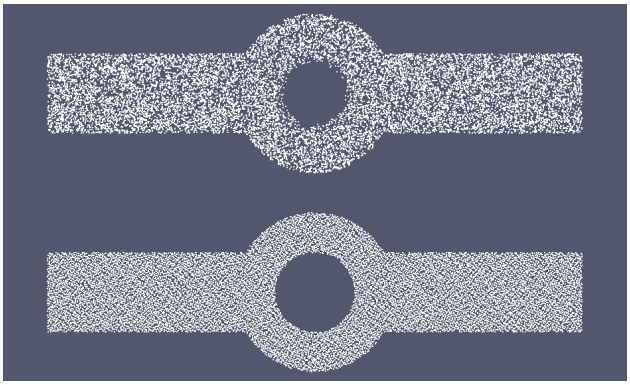


Figure 11: A snapshot of a batch of training points generated using uniform sampling (top) and Halton sequences (bottom) for the annular ring example.

- Zero normal gradient for physical variables with even symmetry.

Details on how to setup an example with symmetry boundary conditions are presented in tutorial 17.

1.6 Advanced Schemes and Architectures

In this section, we discuss some of the advanced and state-of-the-art Deep learning architectures and schemes that have become a part of Modulus library.

1.6.1 Network Architectures in Modulus

1.6.1.1 Fourier Network Neural networks are generally biased toward low-frequency solutions, a phenomenon that is known as "spectral bias" [7]. This can adversely affect the training convergence as well as the accuracy of the model. One approach to alleviate this issue is to perform input encoding, that is, to transform the inputs to a higher-dimensional feature space via high-frequency functions [8, 7, 9]. This is done in Modulus using the Fourier networks, which takes the following form:

$$u_{net}(\mathbf{x};\theta) = \mathbf{W}_n\big\{\phi_{n-1}\circ\phi_{n-2}\circ\cdots\circ\phi_1\circ\phi_E\big\}(\mathbf{x}) + \mathbf{b}_n, \quad \phi_i(\mathbf{x}_i) = \sigma\left(\mathbf{W}_i\mathbf{x}_i + \mathbf{b}_i\right), \tag{20}$$

where $u_{net}(\mathbf{x};\theta)$ is the approximate solution, $\mathbf{x}\in\mathbb{R}^{d_0}$ is the input to network, $\phi_i\in\mathbb{R}^{d_i}$ is the i^{th} layer of the network, $\mathbf{W}_i\in\mathbb{R}^{d_i\times d_{i-1}}, \mathbf{b}_i\in\mathbb{R}^{d_i}$ are the weight and bias of the i^{th} layer, θ denotes the set of network's trainable parameters, i.e., $\theta=\{\mathbf{W}_1,\mathbf{b}_1,\cdots,\mathbf{b}_n,\mathbf{W}_n\}$, n is the number of layers, and σ is the activation function. ϕ_E is an input encoding layer, and by setting that to identity function, we arrive at the standard feed-forward fully-connected architecture. The input encoding layer in Modulus is a variation of the one proposed in [9] with trainable encoding, and takes the following form

$$\phi_E = \left[\sin\left(2\pi\mathbf{f}\times\mathbf{x}\right);\cos\left(2\pi\mathbf{f}\times\mathbf{x}\right)\right]^T, \tag{21}$$

where $\mathbf{f}\in\mathbb{R}^{n_f\times d_0}$ is the trainable frequency matrix and n_f is the number of frequency sets.

In the case of parameterized examples, it is also possible to apply encoding to the parameters in addition to the spatial inputs. In fact, it has been observed that applying encoding to the parametric inputs in addition to the spatial inputs will improve the accuracy and the training convergence of the model. Note that Modulus applies the input encoding to the spatial and parametric inputs in a fully-decoupled setting and then concatenates the spatial/temporal and parametric Fourier features together. Details on the usage of Fourier net can be found Section 12.2 while its application for FPGA heat sink can be found in tutorial 17.

1.6.1.2 Modified Fourier Network In Fourier network, a standard fully-connected neural network is used as the nonlinear mapping between the Fourier features and the model output. In modified Fourier networks, we use a variant of the fully-connected network similar to the one proposed in [10]. Two transformation layers are introduced to project the Fourier features to another learned feature space, and are then used to update the hidden layers through element-wise multiplications, similar to its standard fully connected counterpart in [10]. It is shown in tutorial 17 that this multiplicative interaction can improve the training convergence and accuracy, although at the cost of slightly increasing the training time per iteration. The hidden layers in this architecture take the following form

$$\phi_i(\mathbf{x}_i) = \left(1-\sigma\left(\mathbf{W}_i\mathbf{x}_i+\mathbf{b}_i\right)\right)\odot\sigma\left(\mathbf{W}_{T_1}\phi_E+\mathbf{b}_{T_1}\right)+\sigma\left(\mathbf{W}_i\mathbf{x}_i+\mathbf{b}_i\right)\odot\sigma\left(\mathbf{W}_{T_2}\phi_E+\mathbf{b}_{T_2}\right), \tag{22}$$

where $i>1$ and $\{\mathbf{W}_{T_1},\mathbf{b}_{T_1}\},\{\mathbf{W}_{T_2},\mathbf{b}_{T_2}\}$ are the parameters for the two transformation layers, and ϕ_E takes the form in equation 21. Details on how to use the modified Fourier networks can be found in Section 12.2 while its application for the FPGA heat sink can be found in tutorial 17.

1.6.1.3 Highway Fourier Network Highway Fourier network is Modulus' another variation of the Fourier network, inspired by the highway networks proposed in [11]. Highway networks consist of adaptive gating units that control the flow of information, and are originally developed to be used in the training of very deep networks [11]. The hidden layers in this network take the following form

$$\phi_i(\mathbf{x}_i) = \sigma\left(\mathbf{W}_i\mathbf{x}_i+\mathbf{b}_i\right)\odot\sigma_s\left(\mathbf{W}_T\phi_E+\mathbf{b}_T\right)+\left(\mathbf{W}_P\mathbf{x}+\mathbf{b}_P\right)\odot\left(1-\sigma_s\left(\mathbf{W}_T\phi_E+\mathbf{b}_T\right)\right). \tag{23}$$

Here, σ_s is the sigmoid activation, $\{\mathbf{W}_T,\mathbf{b}_T\}$ are the parameters of the transformer layer, and $\{\mathbf{W}_P,\mathbf{b}_P\}$ are the parameters of the projector layer, which basically projects the network inputs to another space to match with the dimensionality of hidden layers. The transformer layer here controls the relative contribution of the network's hidden state and the network's input to the output of the hidden layer. It also offers a multiplicative interaction mechanism between the network's input and hidden states, similar to the modified Fourier network. Details on how to use the highway Fourier networks can be found in Section 12.2 while its application for the FPGA heat sink can be found in tutorial 17.

1.6.1.4 SiReNs In [12], the authors propose a neural network using Sin activation functions dubbed sinusoidal representation networks or SiReNs. This network has similarities to the Fourier networks above because using a Sin activation function has the same effect as the input encoding for the first layer of the network. A key component of this network architecture is the initialization scheme. The weight matrices of the network are drawn from a uniform distribution $W \sim U(-\sqrt{\frac{6}{fan\_in}}, \sqrt{\frac{6}{fan\_in}})$ where fan_inis is the input size to that layer. The input of each Sin activation has a Gauss-Normal distribution and the output of each Sin activation, an arcSin distribution. This preserves the distribution of activations allowing deep architectures to be constructed and trained effectively [12]. The first layer of the network is scaled by a factor ω to span multiple periods of the Sin function. This was empirically shown to give good performance and is in line with the benefits of the input encoding in the Fourier network. The authors suggest $\omega = 30$ to perform well under many circumstances and is the default value given in Modulus as well. Details on how to use the SiReN architecture in Modulus can be found in Section 12.2.

1.6.1.5 DGM architecture The DGM architecture is proposed by [13], and consists of several fully-connected layers each of which includes a number of sub-layers, similar in spirit to the LSTM architecture, as follows:

$$
\begin{aligned}
&S^1 = \sigma(XW^1 + b^1), \\
&Z^\ell = \sigma(XV_z^\ell + S^\ell W_z^\ell + b_z^\ell), \quad \forall \ell \in \{1, \cdots, n_\ell\}, \\
&G^\ell = \sigma(XV_g^\ell + S^\ell W_g^\ell + b_g^\ell), \quad \forall \ell \in \{1, \cdots, n_\ell\}, \\
&R^\ell = \sigma(XV_r^\ell + S^\ell W_r^\ell + b_r^\ell), \quad \forall \ell \in \{1, \cdots, n_\ell\}, \\
&H^\ell = \sigma(XV_h^\ell + (S^\ell \odot R^\ell)^\ell W_h^\ell + b_h^\ell), \quad \forall \ell \in \{1, \cdots, n_\ell\}, \\
&S^{\ell+1} = (1 - G^\ell) \odot H^\ell + Z^\ell \odot S^\ell, \\
&u_{net}(X; \theta) = S^{n_\ell+1} W + b.
\end{aligned}
\tag{24}
$$

The set of DGM network parameters include

$$
\theta = \{W^1, b^1, \left(V_z^\ell, W_z^\ell, b_z^\ell\right)_{\ell=1}^{n_\ell}, \left(V_g^\ell, W_g^\ell, b_g^\ell\right)_{\ell=1}^{n_\ell}, \left(V_r^\ell, W_r^\ell, b_r^\ell\right)_{\ell=1}^{n_\ell}, \left(V_h^\ell, W_h^\ell, b_h^\ell\right)_{\ell=1}^{n_\ell}, W, b\}.
$$

where X is the input to the network, $\sigma(\cdot)$ is the activation function, n_ℓ is the number of hidden layers, $\odot$ is the Hadamard product, and $u_{net}(X; \theta)$ is the network output. One important feature of this architecture is that it consists of multiple element-wise multiplication of nonlinear transformations of the input, and that can potentially help with learning complicated functions [13]. Details on the usage of the DGM architecture can be found in Section 12.2. Application for this architecture using the FPGA heat sink can be found in tutorial 17.

1.6.1.6 Multiplicative Filter Network Multiplicative filter networks [14] consist of linear or nonlinear transformations of Fourier or Gabor filters of the input, multiplied together at each hidden layer, as follows:

$$
\begin{aligned}
&\phi_1 = f(\mathbf{x}, \xi_1), \\
&\phi_{i+1} = \sigma\left(\mathbf{W}_i \phi_i + \mathbf{b}_i\right) \odot f(\mathbf{x}, \xi_{i+1}), \quad \forall i \in \{1, \cdots, n-1\}, \\
&u_{net}(\mathbf{x}; \theta) = \mathbf{W}_n \phi_n + \mathbf{b}_n.
\end{aligned}
\tag{25}
$$

Here, $f(\mathbf{x}, \xi_i)$ is a multiplicative Fourier or Gabor filter. The set of multiplicative filter network parameters are $\theta = \{\mathbf{W}_1, \mathbf{b}_1, \xi_1, \cdots \mathbf{W}_n, \mathbf{b}_n, \xi_n\}$. Note that in the original implementation in [14], no activation function is used, and network nonlinearity comes from the multiplicative filters only. In this setting, it has been shown in [14] that the output of a multiplicative Filter network can be represented as a linear combination of Fourier or Gabor bases. In Modulus, the user can choose whether to use activation functions or not. The Fourier filters take the following form:

$$
f(\mathbf{x}, \xi_i) = \sin(\omega_i \mathbf{x} + \phi_i), \tag{26}
$$

where $\xi_i = \{\omega_i, \phi_i\}$. The Gabor filters also take the following form:

$$
f(\mathbf{x}, \xi_i) = \exp\left(-\frac{\gamma_i}{2} ||\mathbf{x} - \mu_i||_2^2\right) \sin(\omega_i \mathbf{x} + \phi_i), \tag{27}
$$

where $\xi_i = \{\gamma_i, \mu_i, \omega_i, \phi_i\}$. For detais on the multiplicative filter networks and network initialization, please refer to [14]. Details on how to use the multiplicative filter networks can be found in Section 12.2.

1.6.2 Other Advanced features in Modulus

1.6.2.1 Learning Rate Annealing

Global learning rate annealing. The predominant approach in the training of PINNs is to represent the initial/boundary constraints as additive penalty terms to the loss function. This is usually done by multiplying a parameter λ to each of these terms to balance out the contribution of each term to the overall loss. However, tuning these parameters manually is not straightforward, and also requires treating these parameters as constants. The idea behind the global learning rate annealing, as proposed in [10], is an automated and adaptive rule for dynamic tuning of these parameters during the training. Let us assume the loss function for a non-transient problem takes the following form

$$L(\theta) = L_{residual}(\theta) + \lambda^{(i)} L_{BC}(\theta), \tag{28}$$

where the superscript (i) represents the training iteration index. Then, at each training iteration, the global learning rate annealing scheme [10] computes the ratio between the gradient statistics for the PDE loss term and the boundary term, as follows

$$\bar{\lambda}^{(i)} = \frac{max\left(\left|\nabla_\theta L_{residual}\left(\theta^{(i)}\right)\right|\right)}{mean\left(\left|\nabla_\theta L_{BC}\left(\theta^{(i)}\right)\right|\right)}. \tag{29}$$

Finally, the annealing parameter $\lambda^{(i)}$ is computed using an exponential moving average as follows

$$\lambda^{(i)} = \alpha\bar{\lambda}^{(i)} + (1-\alpha)\lambda^{(i-1)}, \tag{30}$$

where α is the exponential moving average decay.

Local learning rate annealing. Local learning rate annealing is a variation of the global learning rate annealing, for which the annealing parameter for each of the network parameters is computed separately. Let us assume that the SGD update rule for a parameter θ_j takes the following form

$$\theta_j^{(i+1)} = \theta_j^{(i)} - \eta^{(i)} \nabla_{\theta_j} L_{residual}\left(\theta^{(i)}\right) - \lambda_j^{(i)} \eta^{(i)} \nabla_{\theta_j} L_{BC}\left(\theta^{(i)}\right), \tag{31}$$

where $\eta^{(i)}$ is the step size at iteration i. $\lambda_j^{(i)}$ is the local annealing parameter for θ_j, and is computed as

$$\bar{\lambda}_j^{(i)} = \frac{\left|\nabla_{\theta_j} L_{residual}\left(\theta^{(i)}\right)\right|}{\left|\nabla_{\theta_j} L_{BC}\left(\theta^{(i)}\right)\right|}, \tag{32}$$

and,

$$\lambda_j^{(i)} = \alpha\bar{\lambda}_j^{(i)} + (1-\alpha)\lambda_j^{(i)}. \tag{33}$$

Details on how to use the global or local learning rate annealing in Modulus are presented in Section 12.5.

1.6.2.2 Homoscedastic task uncertainty for loss weighting In [15], the authors have proposed to use a Gaussian likelihood with homoscedastic task uncertainty as the training loss in multi-task learning applications. In this scheme, the loss function takes the following form

$$L(\theta) = \sum_{j=1}^{T} \frac{1}{2\sigma_j^2} L_j(\theta) + \log \Pi_{j=1}^{T} \sigma_j, \tag{34}$$

where T is the total number of tasks (or residual and initial/boundary condition loss terms). Minimizing this loss is equivalent to maximizing the log Gaussian likelihood with homoscedastic uncertainty [15], and the uncertainty terms σ serve as adaptive wrights for different loss terms. Figure 12 presents a comparison between the uncertainty loss weighting and no loss weighting for the annular ring example, showing that uncertainty loss weighting improves the training convergence and accuracy in this example. For details on this scheme, please refer to [15].

1.6.2.3 Learning Rate Schedules for Multi-GPU Simulations One common issue when training neural networks with a large number of trainable parameters is the amount of time it takes for the model to converge. An effective way of reducing the time to convergence is to parallelize the training process across multiple GPUs. The most common multi-GPU parallelization strategy is data parallelism where a given global training batch is split into multiple a sub-batches for each GPU. Each GPU then performs the forward and backward passes for its sub-batch and the

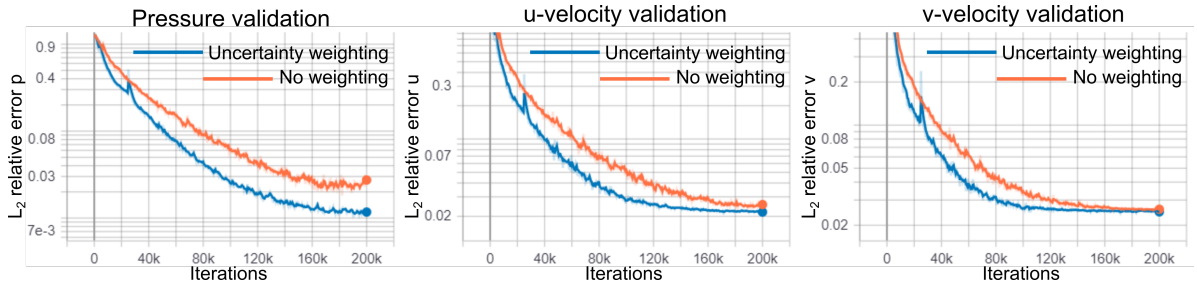


Figure 12: A comparison between the uncertainty loss weighting vs. no loss weighting for the annular ring example.

gradients are accumulated across all the GPUs using an allreduce algorithm. This form of data parallelism is the most computationally efficient when the batch size per GPU is kept constant instead of the global batch size.

It was shown in [16] that the total time to convergence can be reduced linearly with the number of GPUs by proportionally increasing the learning rate. However, doing that naively would cause the model to diverge since the initial learning rate can be very high. An effective solution for this is to have an initial warmup period when the learning rate gradually increases from the baseline to the scaled learning rate. Modulus implements a constant and a linear learning rate warmup scheme in conjunction with an exponential decay baseline learning rate schedule.

For the linear warmup scheme, the learning rate α at step s when run with n GPUs is given by

$$\alpha = \min\left\{\alpha_w(n), \alpha_b\right\} \tag{35}$$

where $\alpha_{0,b}$ is the baseline exponential decay learning rate schedule given by

$$\alpha_b = \alpha_{0,b} r^{s/s_d} + \alpha_{1,b} \tag{36}$$

and $\alpha_w(n)$ is the warmup learning rate given by

$$\alpha_w(n) = \alpha_{0,b}\left[1 + (n-1)\frac{s}{s_w}\right]. \tag{37}$$

Here, s_d is the baseline learning rate decay steps, r is the decay rate, $\alpha_{0,b}$ and $\alpha_{1,b}$ determine the start and end learning rates and s_w is the number of warmup steps.

Details on how to train with multiple GPUs and use the aforementioned learning rate schedule to reduce the time to convergence are presented in Section 19.4.1.

1.6.2.4 Adaptive Activation Functions In training of neural networks, in addition to leaning the linear transformations, one can also learn the nonlinear transformations to potentially improve the convergence as well as the accuracy of the model by using the global adaptive activation functions proposed by [17]. Global adaptive activations consist of a single trainable parameter that is multiplied by the input to the activations in order to modify the slope of activations. Therefore, a nonlinear transformation at layer ℓ will take the following form

$$\mathcal{N}^\ell\left(H^{\ell-1};\theta,a\right) = \sigma\left(a\mathcal{L}^\ell\left(H^{\ell-1}\right)\right), \tag{38}$$

where $\mathcal{N}^\ell$ is the nonlinear transformation at layer ℓ, $H^{\ell-1}$ is the output of the hidden layer $\ell-1$, θ is the set of model weights and biases, a is the global adaptive activation parameter, σ is the activation function, and $\mathcal{L}^\ell$ is the linear transformation at layer ℓ. Similar to the network weights and biases, the global adaptive activation parameter a is also a trainable parameter, and these trainable parameters are optimized by

$$\theta^*, a^* = \arg\min_{\theta,a} L(\theta, a). \tag{39}$$

Details on how to use the global adaptive activations in Modulus are presented in Section 12.3.

1.7 Weak solution of PDEs using PINNs

In previous discussions on PINNs, we aimed at solving the classical solution of the PDEs. However, some physics have no classical (or strong) form but only a variational (or weak) form [18]. This requires handling the PDEs in a different approach other than its original (classical) form, especially for interface problem, concave domain, singular problem, etc. In Modulus, we can solve the PDEs not only in its classical form, but also in it weak form. Before describing the theory for weak solutions of PDEs using PINNs, let's start by the definitions of classical, strong and weak solutions.

Note: The mathematical definitions of the different spaces that are used in the subsequent sections like the L^p, C^k, $W^{k,p}$, H, etc. can be found in the Appendix. For general theory of the partial differential equations and variational approach, we recommend [19, 20].

Classical solution, Strong solution, Weak solution In this section, we introduce the classical solution, strong solution, and weak solution for the Dirichlet problem. Let us consider the following Poisson's equation.

$$\begin{cases} -\Delta u = f & \text{in } \Omega \quad (40) \\ u = 0 & \text{on } \partial\Omega \quad (41) \end{cases}$$

Definition 1.1 (Classical Solution) *Let $f \in C(\overline{\Omega})$ in (40)-(41), then there is a unique solution $u \in C^2(\Omega) \cap C_0^1(\Omega)$ for (40)-(41). We call this solution as the classical solution of (40)-(41).*

Definition 1.2 (Strong Solution) *Let $f \in L^2(\Omega)$ in (40)-(41), then there is a unique solution $u \in H^2(\Omega) \cap H_0^1(\Omega)$ for (40)-(41). We call this solution as the strong solution of (40)-(41).*

From the definition of strong solution and Sobolev space, we can see that the solution of (40)-(41) is actually the solution of the following problem: Finding a $u \in H^2(\Omega) \cap H_0^1(\Omega)$, such that

$$\int_\Omega (\Delta u + f) v dx = 0 \qquad \forall v \in C_0^\infty(\Omega) \tag{42}$$

By applying integration by parts and (41), we get

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx$$

This leads us to the definition of weak solution as the following.

Definition 1.3 (Weak Solution) *Let $f \in L^2(\Omega)$ in (40)-(41), then there is a unique solution $u \in H_0^1(\Omega)$ for the following problem: Finding a $u \in H_0^1(\Omega)$ such that*

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx \qquad \forall v \in H_0^1(\Omega). \tag{43}$$

We call this solution as the weak solution of (40)-(41).

In simpler terms, the difference between these three types of solutions can be summarized as below:

Remark 1.1 *The essential difference among classical solution, strong solution and weak solution is their regularity requirements. The classic solution is a solution with 2nd order continuous derivatives. The strong solution has 2nd order weak derivatives, while the weak solution has weak 1st order weak derivatives. Obviously, classical solution has highest regularity requirement and the weak solution has lowest one.*

PINNs for obtaining weak solution Now we will discuss how PINNs can be used to handle the PDEs in approaches different than its original (classical) form. In [21, 22], the authors introduced the VPINN and hp-VPINN methods to solve PDEs' integral form. This integral form is based on (42). Hence, it is solving a strong solution, which is better than a classical solution.

To further improve the performance of PINNs, we establish the method based on (43) i.e., we are solving the weak solution. Let us assume we are solving (40)-(41). To seek the weak solution, we may focus on the following variational form:

$$\begin{aligned} \int_\Omega \nabla u \cdot \nabla v dx &= \int_\Omega f v dx \quad (44) \\ u &= 0 \quad \text{on } \partial\Omega \quad (45) \end{aligned}$$

For (45), we may handle it as the traditional PINNs: take random points $\{\mathbf{x}_i^b\}_{i=1}^{N_b} \subset \partial\Omega$, then the boundary loss is

$$MSE_b = \frac{1}{N_b}\sum_{i=1}^{N_b}\left(u_{NN}(\mathbf{x}_i^b) - 0\right)^2$$

For (44), we choose a quadrature rule $\{\mathbf{x}_i^q, w_i^q\}_{i=1}^{N_q}$, such that for $u : \Omega \mapsto \mathbb{R}$, we have

$$\int_\Omega u dx \approx \sum_{i=1}^{N_q} w_i^q u(\mathbf{x}_i^q).$$

For uniform random points or quasi Monte Carlo points, $w_i^q = 1/N_q$ for $i = 1, \cdots, N_q$. Additionally, we choose a set of test functions $v_j \in V_h$, $j = 1, \cdots, M$ and then the loss of the integral is

$$MSE_v = \left[\sum_{i=1}^{N_q} w_i^q \left(\nabla u(\mathbf{x}_i^q)\cdot\nabla v_j(\mathbf{x}_i^q) - f(\mathbf{x}_i^q)v_j(\mathbf{x}_i^q)\right)\right]^2.$$

Then, the total loss is

$$MSE = \lambda_v * MSE_v + \lambda_b * MSE_b,$$

where the λ_v and λ_b are the corresponding weights for each terms.

As we will see in the tutorial example 9, this scheme is flexible and can handle the interface and Neumann boundary condition easily. We can also use more than one neural networks on different domains by applying the discontinuous Galerkin scheme.

1.8 Generalized Polynomial Chaos

In this section, we briefly introduce the generalized Polynomial Chaos (gPC) expansion, an efficient method for assessing how the uncertainties in a model input manifest in its output. Later in Section 18.5, we show how the gPC can be used as a surrogate for shape parameterization in both of the tessellated and constructive solid geometry modules.

The pth-degree gPC expansion for a d-dimensional input $\boldsymbol{\Xi}$ takes the following form

$$u_p(\boldsymbol{\Xi}) = \sum_{\mathbf{i} \in \Lambda_{p,d}} c_{\mathbf{i}} \psi_{\mathbf{i}}(\boldsymbol{\Xi}), \tag{46}$$

where $\mathbf{i}$ is a multi-index and $\Lambda_{p,d}$ is the set of multi-indices defined as

$$\Lambda_{p,d} = \{\mathbf{i} \in \mathbb{N}_0^d : ||\mathbf{i}||_1 \leq p\}, \tag{47}$$

and the cardinality of $\Lambda_{d,p}$ is

$$C = |\Lambda_{p,d}| = \frac{(p+d)!}{p!d!}. \tag{48}$$

$\{c_{\mathbf{i}}\}_{\mathbf{i} \in \mathbb{N}_0^d}$ is the set of unknown coefficients of the expansion, and can be determined based on the methods of stochastic Galerkin, stochastic collocation, or least square [23]. For the example presented in this user guide, we will use the least square method. Although the number of required samples to solve this least square problem is C, it is recommended to use at least $2C$ samples for a reasonable accuracy [23]. $\{\psi_{\mathbf{i}}\}_{\mathbf{i} \in \mathbb{N}_0^d}$ is the set of orthonormal basis functions that satisfy the following condition

$$\int \psi_{\mathbf{m}}(\xi)\psi_{\mathbf{n}}(\xi)\rho(\xi)d\xi = \delta_{\mathbf{mn}}, \quad \mathbf{m}, \mathbf{n} \in \mathbb{N}_0^d. \tag{49}$$

For instance, for a uniformly and normally distributed ψ, the normalized Legendre and Hermite polynomials, respectively, satisfy the orthonormality condition in Equation 49.

Appendices

A Relative Function Spaces and Integral Identities

In this section, we give some essential definitions of Relative function spaces, Sobolev spaces and some important equalities. All the integral in this section should be understood by Lebesgue integral.

A.1 L^p space

Let $\Omega \subset \mathbb{R}^d$ is an open set. For any real number $1 < p < \infty$, we define

$$L^p(\Omega) = \left\{ u : \Omega \mapsto \mathbb{R} \middle| u \text{ is measurable on } \Omega, \int_\Omega |u|^p dx < \infty \right\},$$

endowed with the norm

$$\|u\|_{L^p(\Omega)} = \left(\int_\Omega |u|^p dx \right)^{\frac{1}{p}}.$$

For $p = \infty$, we have

$$L^\infty(\Omega) = \left\{ u : \Omega \mapsto \mathbb{R} \middle| u \text{ is uniformly bounded in } \Omega \text{ except a zero measure set} \right\},$$

endowed with the norm

$$\|u\|_{L^\infty(\Omega)} = \sup_\Omega |u|.$$

Sometimes, for functions on unbounded domains, we consider their local integrabilities. To this end, we define the following local L^p space

$$L^p_{loc}(\Omega) = \left\{ u : \Omega \mapsto \mathbb{R} \middle| u \in L^p(V),\ \forall V \subset\subset \Omega \right\},$$

where $V \subset\subset \Omega$ means V is a compact subset of Ω.

A.2 C^k space

Let $k \geq 0$ be an integer, and $\Omega \subset \mathbb{R}^d$ is an open set. The $C^k(\Omega)$ is the k-th differentiable function space given by

$$C^k(\Omega) = \left\{ u : \Omega \mapsto \mathbb{R} \middle| u \text{ is } k\text{-times continuously differentiable} \right\}.$$

Let $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \cdots, \alpha_d)$ be a d-fold multiindex of order $|\boldsymbol{\alpha}| = \alpha_1 + \alpha_2 + \cdots + \alpha_n = k$. The k-th order (classical) derivative of u is denoted by

$$D^{\boldsymbol{\alpha}} u = \frac{\partial^k}{\partial x_1^{\alpha_1} \partial x_2^{\alpha_2} \cdots \partial x_d^{\alpha_d}} u.$$

For the closure of Ω, denoted by $\overline{\Omega}$, we have

$$C^k(\overline{\Omega}) = \left\{ u : \Omega \mapsto \mathbb{R} \middle| D^{\boldsymbol{\alpha}} u \text{ is uniformly continuous on bounded subsets of } \Omega, \forall |\boldsymbol{\alpha}| \leq k \right\}.$$

When $k = 0$, we also write $C(\Omega) = C^0(\Omega)$ and $C(\overline{\Omega}) = C^0(\overline{\Omega})$.

We also define the infinitely differentiable function space

$$C^\infty(\Omega) = \left\{ u : \Omega \mapsto \mathbb{R} \middle| u \text{ is infinitely differentiable} \right\} = \bigcap_{k=0}^{\infty} C^k(\Omega)$$

and

$$C^\infty(\overline{\Omega}) = \bigcap_{k=0}^{\infty} C^k(\overline{\Omega}).$$

We use $C_0(\Omega)$ and $C_0^k(\Omega)$ denote these functions in $C(\Omega)$, $C^k(\Omega)$ with compact support.

A.3 $W^{k,p}$ space

The weak derivative is given by the following definition [20].

Definition A.1 *Suppose* $u,\ v \in L^1_{loc}(\Omega)$ *and* $\boldsymbol{\alpha}$ *is a multiindex. We say that* v *is the* $\boldsymbol{\alpha}^{th}$ *weak derivative of* u*, written*

$$D^{\boldsymbol{\alpha}} u = v,$$

provided

$$\int_\Omega u D^{\boldsymbol{\alpha}} \phi dx = (-1)^{|\boldsymbol{\alpha}|} \int_\Omega v \phi dx$$

for all test functions $\phi \in C_0^\infty(\Omega)$.

As a typical example, let $u(x) = |x|$ and $\Omega = (-1, 1)$. For calculus we know that u is not (classical) differentiable at $x = 0$. However, it has weak derivative

$$(Du)(x) = \begin{cases} 1 & x > 0, \\ -1 & x \leq 0. \end{cases}$$

Definition A.2 *For an integer* $k \geq 0$ *and real number* $p \geq 1$*, the Sobolev space is defined by*

$$W^{k,p}(\Omega) = \left\{ u \in L^p(\Omega) \middle| D^{\boldsymbol{\alpha}} u \in L^p(\Omega),\ \forall |\boldsymbol{\alpha}| \leq k \right\},$$

endowed with the norm

$$\|u\|_{k,p} = \left(\int_\Omega \sum_{|\boldsymbol{\alpha}| \leq k} |D^{\boldsymbol{\alpha}} u|^p \right)^{\frac{1}{p}}.$$

Obviously, when $k = 0$, we have $W^{0,p}(\Omega) = L^p(\Omega)$.

When $p = 2$, $W^{k,p}(\Omega)$ is a Hilbert space. And it also denoted by $H^k(\Omega) = W^{k,2}(\Omega)$. The inner product in $H^k(\Omega)$ is given by

$$\langle u, v \rangle = \int_\Omega \sum_{|\boldsymbol{\alpha}| \leq k} D^{\boldsymbol{\alpha}} u D^{\boldsymbol{\alpha}} v dx$$

A crucial subset of $W^{k,p}(\Omega)$, denoted by $W_0^{k,p}(\Omega)$, is

$$W_0^{k,p}(\Omega) = \left\{ u \in W^{k,p}(\Omega) \middle| D^{\boldsymbol{\alpha}} u|_{\partial\Omega} = 0,\ \forall |\boldsymbol{\alpha}| \leq k - 1 \right\}.$$

It is customary to write $H_0^k(\Omega) = W_0^{k,2}(\Omega)$.

A.4 Integral identities

In this subsection, we assume $\Omega \subset \mathbb{R}^d$ is a Lipschitz bounded domain (see [24] for the definition of Lipschitz domain).

Theorem A.1 (Green's formulas) *Let* $u,\ v \in C^2(\overline{\Omega})$*. Then*

1. $$\int_\Omega \Delta u dx = \int_{\partial\Omega} \frac{\partial u}{\partial n} dS$$

2. $$\int_\Omega \nabla u \cdot \nabla v dx = -\int_\Omega u \Delta v dx + \int_{\partial\Omega} u \frac{\partial v}{\partial n} dS$$

3. $$\int_\Omega u \Delta v - v \Delta u dx = \int_{\partial\Omega} u \frac{\partial v}{\partial n} - v \frac{\partial u}{\partial n} dS$$

For curl operator we have some similar identities. To begin with, we define the 1D and 2D curl operators. For a scalar function $u(x_1, x_2) \in C^1(\overline{\Omega})$, we have

$$\nabla \times u = \left(\frac{\partial u}{\partial x_2}, -\frac{\partial u}{\partial x_1} \right)$$

For a 2D vector function $\mathbf{v} = (v_1(x_1, x_2), v_2(x_1, x_2)) \in (C^1(\overline{\Omega}))^2$, we have

$$\nabla \times \mathbf{v} = \frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2}$$

Then we have the following integral identities for curl operators.

Theorem A.2 *1. Let $\Omega \subset \mathbb{R}^3$ and $\mathbf{u},\ \mathbf{v} \in (C^1(\overline{\Omega}))^3$. Then*

$$\int_\Omega \nabla \times \mathbf{u} \cdot \mathbf{v} dx = \int_\Omega \mathbf{u} \cdot \nabla \times \mathbf{v} dx + \int_{\partial\Omega} \mathbf{n} \times \mathbf{u} \cdot \mathbf{v} dS,$$

where $\mathbf{n}$ is the $\mathbf{n}$ is the unit outward normal.

2. Let $\Omega \subset \mathbb{R}^2$ and $\mathbf{u} \in (C^1(\overline{\Omega}))^2$ and $v \in C^1(\overline{\Omega})$. Then

$$\int_\Omega \nabla \times \mathbf{u} v dx = \int_\Omega \mathbf{u} \cdot \nabla \times v dx + \int_{\partial\Omega} \boldsymbol{\tau} \cdot \mathbf{u} v dS,$$

where $\boldsymbol{\tau}$ is the unit tangent to $\partial\Omega$.

B Example: Derivation of Variational form for the interface problem

Let $\Omega_1 = (0, 0.5) \times (0, 1)$, $\Omega_2 = (0.5, 1) \times (0, 1)$, $\Omega = (0, 1)^2$. The interface is $\Gamma = \overline{\Omega}_1 \cap \overline{\Omega}_2$, and the Dirichlet boundary is $\Gamma_D = \partial\Omega$. The domain for the problem can be visualized in the Figure 13. The problem was originally defined in [25].

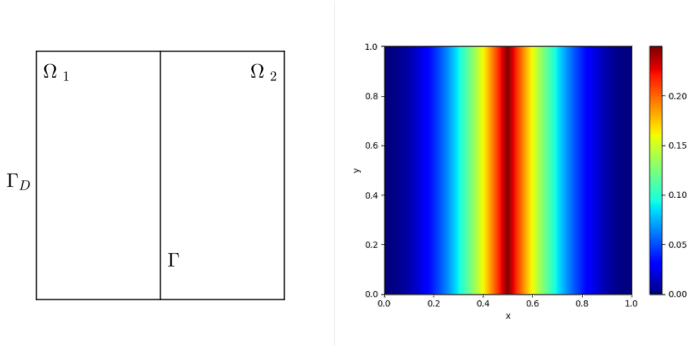


Figure 13: Left: Domain of interface problem. Right: True Solution

The PDEs for the problem are defined as

$$\begin{cases} -\Delta u = f & \text{in } \Omega_1 \cup \Omega_2, \quad (50) \\ u = g_D & \text{on } \Gamma_D, \quad (51) \\ \left[\dfrac{\partial u}{\partial \mathbf{n}} \right] = g_I & \text{on } \Gamma, \quad (52) \end{cases}$$

where $f = -2$, $g_I = 2$ and

$$g_D = \begin{cases} x^2 & 0 \le x \le \frac{1}{2} \\ (x-1)^2 & \frac{1}{2} < x \le 1 \end{cases}.$$

The g_D is the exact solution of (50)-(52).

The jump $[\cdot]$ on the interface Γ is defined by

$$\left[\frac{\partial u}{\partial \mathbf{n}} \right] = \nabla u_1 \cdot \mathbf{n}_1 + \nabla u_2 \cdot \mathbf{n}_2, \quad (53)$$

where u_i is the solution in Ω_i and the $\mathbf{n}_i$ is the unit normal on $\partial\Omega_i \cap \Gamma$.

As suggested in the original reference, this problem does not accept a strong (classical) solution but only a unique weak solution (g_D) which is shown in Figure 13.

Note: It is noted that in the original paper [25], the PDE is incorrect and (50)-(52) defines the corrected PDEs for the problem.

We now construct the variational form of (50)-(52). This is the first step to obtain its weak solution. Since (52) suggests that the solution's derivative is broken at interface (Γ), we have to do the variational form on Ω_1 and Ω_2 separately. Specifically, let v_i be a suitable test function on Ω_i, and by integration by parts, we have for $i = 1, 2$,

$$\int_{\Omega_i} (\nabla u \cdot \nabla v_i - f v_i) dx - \int_{\partial\Omega_i} \frac{\partial u}{\partial \mathbf{n}} v_i ds = 0.$$

If we are using one neural network and a test function defined on whole Ω, then by adding these two equalities, we have

$$\int_{\Omega} (\nabla u \cdot \nabla v - f v) dx - \int_{\Gamma} g_I v ds - \int_{\Gamma_D} \frac{\partial u}{\partial \mathbf{n}} v ds = 0 \tag{54}$$

If we are using two neural networks, and the test functions are different on Ω_1 and Ω_2, then we may use the discontinuous Galerkin formulation [26]. To this end, we first define the jump and average of scalar and vector functions. Consider the two adjacent elements as shown in Figure 14. $\mathbf{n}^+$ and $\mathbf{n}^-$and unit normals for T^+, T^- on $F = T^+ \cap T^-$, respectively. As we can observe, we have $\mathbf{n}^+ = -\mathbf{n}^-$.

Let u^+ and u^- be two scalar functions on T^+ and T^-, and $\mathbf{v}^+$ and $\mathbf{v}^-$ are two vector fields on T^+ and T^-, respectively. The jump and the average on F is defined by

$$\langle u \rangle = \frac{1}{2}(u^+ + u^-) \qquad \langle \mathbf{v} \rangle = \frac{1}{2}(\mathbf{v}^+ + \mathbf{v}^-)$$
$$[\![u]\!] = u^+ \mathbf{n}^+ + u^- \mathbf{n}^- \qquad [\![\mathbf{v}]\!] = \mathbf{v}^+ \cdot \mathbf{n}^+ + \mathbf{v}^- \cdot \mathbf{n}^-$$

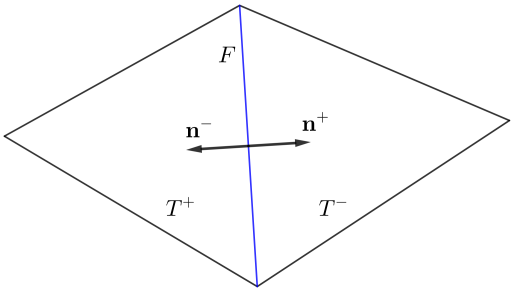


Figure 14: Adjacent Elements.

Lemma B.1 *On F of Figure 14, we have*

$$[\![u\mathbf{v}]\!] = [\![u]\!]\langle \mathbf{v} \rangle + [\![\mathbf{v}]\!]\langle u \rangle.$$

By using the above lemma, we have the following equality, which is an essential tool for discontinuous formulation.

Theorem B.1 *Suppose Ω has been partitioned into a mesh. Let $\mathcal{T}$ be the set of all elements of the mesh, $\mathcal{F}_I$ be the set of all interior facets of the mesh, and $\mathcal{F}_E$ be the set of all exterior (boundary) facets of the mesh. Then we have*

$$\sum_{T \in \mathcal{T}} \int_{\partial T} \frac{\partial u}{\partial \mathbf{n}} v ds = \sum_{e \in \mathcal{F}_I} \int_e \left([\![\nabla u]\!] \langle v \rangle + \langle \nabla u \rangle [\![v]\!] \right) ds + \sum_{e \in \mathcal{F}_E} \int_e \frac{\partial u}{\partial \mathbf{n}} v ds \tag{55}$$

Using (53) and (55), we have the following variational form

$$\sum_{i=1}^{2} (\nabla u_i \cdot v_i - f v_i) dx - \sum_{i=1}^{2} \int_{\Gamma_D} \frac{\partial u_i}{\partial \mathbf{n}} v_i ds - \int_{\Gamma} (g_I \langle v \rangle + \langle \nabla u \rangle [\![v]\!]) ds = 0 \tag{56}$$

Details on how to use these forms can be found in tutorial 9.

2 Lid Driven Cavity Flow

2.1 Introduction

In this tutorial, we will walk through the process of generating a 2D flow simulation for the Lid Driven Cavity (LDC) flow using Modulus. In this tutorial, you would learn the following:

1. How to generate a 2D geometry using the geometry module in Modulus.
2. How to set up the boundary conditions.
3. How to select the flow equations to be solved.
4. How to interpret the different losses and tuning the network.
5. How to do some basic post-processing using Modulus.

Prerequisites

To start with this tutorial, you should have a basic understanding of PDEs, CFD, and Python programming. It is also recommended that you go through the Theory chapter and code documentation before starting. The tutorial also assumes that you have successfully downloaded the Modulus repository.

2.2 Problem Description

The geometry for the problem is shown in figure 15. All the boundaries of the geometry are stationary walls with the top wall moving in the x-direction at a velocity of 1 m/s. The Reynolds number based on the cavity height is chosen to be 10.

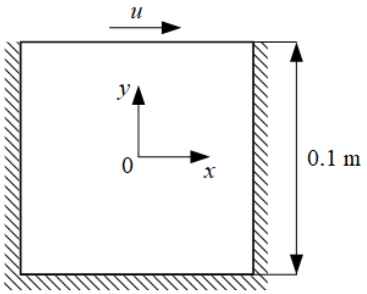


Figure 15: Lid driven cavity geometry

2.3 Case Setup

To set up a case in Modulus, you can create a single python script containing the geometry, boundary conditions, and network information. Once the Modulus repository has been downloaded, you can create a directory and make a .py file using any text editor of your choice.

Before we jump into the case setup in Modulus, lets try to summarize the concepts discussed in the Theory chapter and how it relates to Modulus' features. For solving any physics simulation that is defined by differential equations, we need information about the domain of the problem and its governing equations/boundary conditions. With this data, one can use a solver to solve the equations over the domain to obtain a solution. In Modulus, we have modules/functions to help in each stage of this problem definition and solution. The domain can be defined using either the Modulus' Constructive Solid Geometry (CSG) module or STL module or even some raw point cloud data from CSV file/numpy arrays. Once we have this geometry/point cloud, it can be sub-sampled into points on the boundaries to satisfy the boundary conditions; and interior regions to minimize the PDE/ODE residuals. Once this definition is established in the `TrainDomain` class, we can define the solver setup in the `Solver` class. Here we will have to define the neural network architecture and define the equations again so that the solver can compute the required forward and backward passes in the backend to formulate the required loss function to train the network. Figure 16 summarizes these concepts.

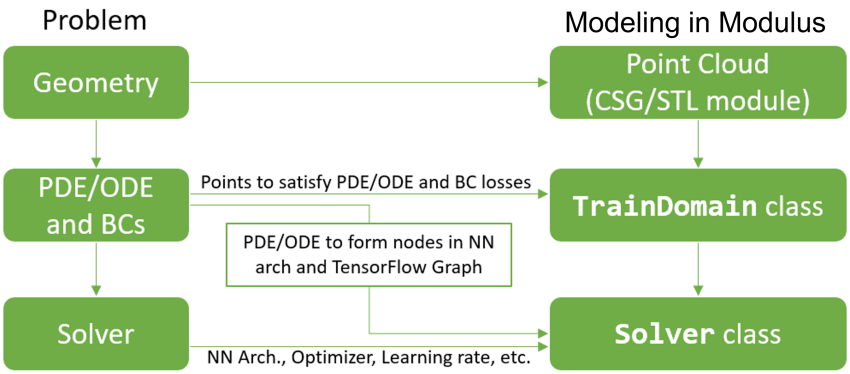


Figure 16: Modulus problem solving methodology

Now with this background, let's walk through the different parts of the code.

Note: The python script for this problem can be found at `examples/ldc/ldc_2d.py`

Importing the required packages

The following part of the code imports all the required packages for creating the geometry, network, and plotting the results.

```
from sympy import Symbol, Eq, Abs

from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain
from modulus.data import Validation
from modulus.sympy_utils.geometry_2d import Rectangle
from modulus.csv_utils.csv_rw import csv_to_dict
from modulus.PDES import NavierStokes
from modulus.controller import ModulusController
```

Listing 1: Importing required packages

2.3.1 Creating Geometry

Once the required packages are imported, we will generate the LDC geometry using the geometry module in Modulus. The geometry module currently supports rectangle, circle, triangle, infinite channel, line in 2D and sphere, cone, cuboid, infinite channel, plane, cylinder, torus, tetrahedron, and triangular prism in 3D. Other complex geometries can be constructed using these primitives by performing operations like add, subtract, intersect, etc. Please refer to the source code documentation for more details on the mode of definition for each shape as well as updates on newly added geometries.

We begin by defining the required variables for the geometry and then generating the 2D square geometry by instantiating an object of `Rectangle` type. In Modulus, a `Rectangle` class is defined using the coordinates for two opposite corner points. The below code shows the process of generating a simple geometry in Modulus.

```
# params for domain
height = 0.1
width = 0.1
vel = 1.0

# define geometry
rec = Rectangle((-width / 2, -height / 2), (width / 2, height / 2))
geo = rec
```

Listing 2: Defining geometry

To visualize the geometry, we can sample either on the boundary or in the interior of the geometry. One such way is shown below where the `sample_boundary` method samples points on the boundary of the geometry. The `sample_boundary` can be replaced by `sample_interior` to sample points in the interior of the geometry.

The var_to_vtk function will generate a .vtu point cloud file for the geometry which can be viewed using tools like Paraview. Alternatively, one can simply do a scatter plot of the samples file.

```
samples = geo.sample_boundary(1000)
var_to_vtk(samples, './geo')
```

Listing 3: Visualizing geometry

Note: The geometry module also features functionality like translate and rotate to generate shapes in arbitrary orientation. We will go through the use of these in upcoming tutorials.

2.3.2 Defining the Boundary conditions and Equations to solve

As described in the Theory chapter, we need to define a training domain for training our Neural Network. A loss function is then constructed which is a combination of contributions from the boundary conditions and equations that a neural network must satisfy at the end of the training. These training points (BCs and equations) are defined in a child class that inherits from the TrainDomain parent class. Currently the boundary conditions are implemented as soft constraints in Modulus. These BCs along with the equations to be solved are used to formulate a composite loss that is minimized by the network during training. For more details, you are encouraged to refer to the Theory chapter.

2.3.2.1 Boundary conditions: For generating a boundary condition in Modulus, we need to sample the points on the required boundary/surface of the geometry and then assign them the desired values.

boundary_bc
A boundary can be sampled using the boundary_bc function. This will, however, sample the entire boundary of the geometry, in this case, all the sides of the rectangle. A particular boundary of the geometry can be sub-sampled by using a particular criterion for the boundary_bc using the criteria parameter. For example, to sample the top wall, criteria is set to y=height/2.

The desired values for the boundary condition are listed as a dictionary in outvar_sympy. In the Modulus framework, we define these variables as keys of this dictionary which are converted to appropriate nodes in the computational graph.

The number of points to sample on each boundary are specified using the batch_size_per_area parameter. The actual number of points sampled is then equal to the perimeter (or) area (or) volume of the geometry being sampled (boundary or interior) times the batch_size_per_area. Eg. In this case, by specifying the batch_size_per_area as 10000 for the top wall, the actual points sampled is 1000, because the length(perimeter) of the top wall is 0.1 m

Note:

- The criteria parameter is optional. With no criteria, boundary_bc will sample on all the boundaries in the geometry.
- The network directory will only show the points sampled in a single batch. However the total points used in the training can be computed by further multiplying the batch size by batch_per_epoch parameter. The default value of this is set to 1000. In the above example, the total points sampled on the Top BC will be 1000 x 1000 = 1000000.

For the LDC problem, we define the top wall with a u velocity equal to 1 m/s in the $+ve$ x-direction while all other walls are stationary (u, v = 0). It can be observed from figure 17 that this gives rise to sharp discontinuities, wherein the u velocity jumps from 0 to 1.0 sharply. As outlined in the tutorial 1, section 1.5.2, we will specify the weighting for this boundary such that the weight of the loss is 0 on the boundaries. We use function $1.0 - 20.0|x|$ as shown in figure 17 for this purpose. Similar to the advantages of weighting losses for equations, (figure 6), eliminating such discontinuities speeds up convergence and allows us to achieve better accuracy.

Weights to any variables can be specified as an input to the lambda_sympy parameter. A keyword 'lambda_' is added onto the key names specified in the outvar_sympy to specify the corresponding weights in the total loss function.

2.3.2.2 Equations to solve: The conservation equations of fluid mechanics are enforced on all the points in the interior of the geometry.

interior_bc
Similar to sampling boundaries, we will use interior_bc function to sample points in the interior of a geometry. The equations to solve are specified as a dictionary input to outvar_sympy.

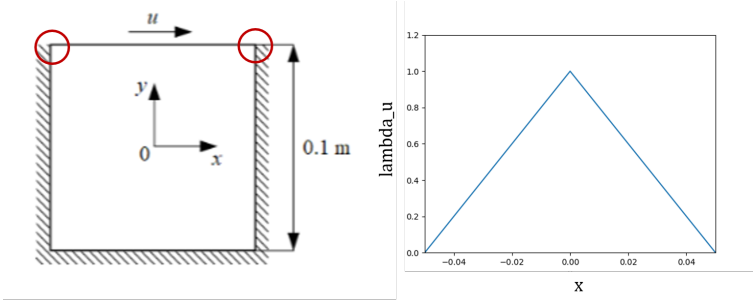


Figure 17: Weighting the sharp discontinuities in the boundary condition

The list of available equations can be found in the PDEs folder. The required equations are then imported at the beginning of the file. Similar to boundaries, we define these equations as keys in the dictionary which acts as input to the `outvar_sympy` parameter. These dictionaries are then used when unrolling the computational graph for training.

For the 2D LDC case, we will solve the continuity equation and the momentum equations in x and y directions. Therefore we will have keys for `'continuity'`, `'momentum_x'` and `'momentum_y'`. We assign the value 0 to these keys as there would be no forcing/source term. If you want to add any source term, the value 0 can be replaced with source value. To see how the equation keys are defined, you can take a look at the Modulus source or refer the source code documentation.

`modulus/PDES/navier_stokes.py`

As an example, the definition of `'continuity'` is presented here.

```
...
# set equations
self.equations = {}
self.equations['continuity'] = rho.diff(t) + (rho*u).diff(x) + (rho*v).diff(y) + (
rho*w).diff(z)
...
```

The resulting in losses are now formulated as depicted in Equations 57, 58, and 59.

$$L_{continuity} = \frac{V}{N}\sum_{i=0}^{N}(0 - continuity(x_i, y_i))^2 \tag{57}$$

$$L_{momentum_x} = \frac{V}{N}\sum_{i=0}^{N}(0 - momentum_x(x_i, y_i))^2 \tag{58}$$

$$L_{momentum_y} = \frac{V}{N}\sum_{i=1}^{n}(0 - momentum_y(x_i, y_i))^2 \tag{59}$$

The parameter `bounds`, determines the range for sampling the values for variables x and y. The `lambda_sympy` parameter is used to determine the weights for different losses. In this problem, we weight each equation at each point by its distance from the boundary by using the Signed Distance Field (SDF) of the geometry. This implies that the points away from the boundary are weighted higher compared to the ones closer to the boundary. We found that this type of weighting of the loss functions leads to a faster convergence since it avoids discontinuities at the boundaries (section 1.5.2).

Note: The `lambda_sympy` parameter is optional. If not specified, the loss for each equation/boundary variable at each point is weighted equally.

```
# define sympy varaibles to parametize domain curves
x, y = Symbol('x'), Symbol('y')

class LDCTrain(TrainDomain):
  def __init__(self, **config):
```

```
    super(LDCTrain, self).__init__()

    #top wall
    topWall = geo.boundary_bc(outvar_sympy={'u': vel, 'v': 0},
                              batch_size_per_area=10000,
                              lambda_sympy={'lambda_u': 1.0 - 20 * Abs(x),  # weight edges to be zero
                                            'lambda_v': 1.0},
                              criteria=Eq(y, height / 2))
    self.add(topWall, name="TopWall")

    # no slip
    bottomWall = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                                 batch_size_per_area=10000,
                                 criteria=y < height / 2)
    self.add(bottomWall, name="NoSlip")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0},
                               bounds={x: (-width / 2, width / 2),
                                       y: (-height / 2, height / 2)},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                                             'lambda_momentum_x': geo.sdf,
                                             'lambda_momentum_y': geo.sdf},
                               batch_size_per_area=400000)
    self.add(interior, name="Interior")
```

Listing 4: Defining boundary conditions and the equations to solve

2.3.3 Creating Validation data

The Modulus solver is a physics based Neural Network solver. Meaning, unlike data driven solvers, it does not require training data from any other CFD/PDE solver to make predictions. Rather it uses Neural Networks as function approximators to solve the required PDEs. More information about this can be found in the Theory chapter.

However, one can add CFD data or data from any other PDE solver and use that to make a comparison with Modulus' results. We will now see how to set up such a validation domain in Modulus. We would use the results from OpenFOAM, an open-source CFD solver to make comparisons. The results can be imported into Modulus as a .csv file. The .csv file is converted to a dictionary of numpy variables for input and output.

Then the validation domain is created by inheriting from the `ValidationDomain` parent class. The dictionary of generated numpy arrays for input and output variables is used as an input to the class method `from_numpy`.

```
# validation data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'U:0': 'u', 'U:1': 'v', 'p': 'p'}
openfoam_var = csv_to_dict('openfoam/cavity_uniformVel0.csv', mapping)
openfoam_var['x'] += -width / 2  # center OpenFoam data
openfoam_var['y'] += -height / 2  # center OpenFoam data
openfoam_invar_numpy = {key: value for key, value in openfoam_var.items() if key in ['x', 'y']}
openfoam_outvar_numpy = {key: value for key, value in openfoam_var.items() if key in ['u', 'v']}

class LDCVal(ValidationDomain):
  def __init__(self, **config):
    super(LDCVal, self).__init__()
    val = Validation.from_numpy(openfoam_invar_numpy, openfoam_outvar_numpy)
    self.add(val, name='Val')
```

Listing 5: Defining the Validation data

2.3.4 Making the Neural Network solver

The solver is defined by inheriting the `Solver` parent class. The `train_domain` and `val_domain` are assigned. The equations to be solved are specified under `self.equations`. The continuity equation is included in the `NavierStokes` equation class. Since this is a 2D problem, `dim` is specified as 2. The kinematic viscosity ν is specified as 0.01 m^2/s and the density ρ is specified as 1.0 kg/m^3.

The inputs and the outputs of the neural network are specified and the nodes of the architecture are made. The default network consists of 6 hidden layers with 512 nodes in each layer. The directory to store the results and the update frequency can be edited in the `update_defaults` function. The default is set to 1000 steps. We will specify the `max_steps` for this simulation to be 400,000 (default is 10,000,000).

```
class LDCSolver(Solver):
```

```
  train_domain = LDCTrain
  val_domain = LDCVal

  def __init__(self, **config):
    super(LDCSolver, self).__init__(**config)
    self.equations = NavierStokes(nu=0.01, rho=1.0, dim=2,
                                  time=False).make_node()
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_ldc_2d',
        'decay_steps': 4000,
        'max_steps': 400000
    })


if __name__ == '__main__':
  ctr = ModulusController(LDCSolver)
  ctr.run()
```

Listing 6: Defining the Neural Network Solver

Awesome! We have just completed the file set up for Modulus. We are now ready to solve the CFD simulation using Modulus' Neural Network solver.

2.4 Running the Modulus solver

Once the python script is set up, we can save the file and exit out of the text editor. In the command prompt, we type in:

```
python ldc_2d.py
```

The console should print the losses at each step. However it is difficult to monitor convergence through the command window and we will use Tensorboard instead, to graphically monitor the losses as the training progresses.

If you want to view all the boundary conditions before actually solving the equations, to ensure correct set up, the script can be run with the following flag:

```
python ldc_2d.py --run_mode=plot_data
```

This should populate the files in the `network_checkpoint_ldc_2d/train_domain/data/` directory. The different `--run_mode` available are:

- `solve`: Default run mode. Trains the neural network.
- `plot_data`: Plots all the domains without training. Useful for debugging BCs, ICs, visualizing domains, point-clouds, etc. before starting the training.
- `eval`: Evaluates all the domains using the last saved training checkpoint. Useful for post-processing where additional domains can be added after training is complete.

2.5 Results and Post-processing

2.5.1 Setting up Tensorboard

Tensorboard is a great tool for visualization of machine learning experiments. More information about tensorboard can be found at: https://www.tensorflow.org/tensorboard. To visualize the various training and validation losses, tensorboard can be set up as follows:

1. In a separate terminal window, we navigate to the location where `network_checkpoint_ldc_2d` directory is saved.
2. In the command line, we type in the following command:

   ```
   tensorboard --logdir=./ --port=7007
   ```

One can specify any desired port they wish. Here we will use `7007`. Once run, the command prompt will display the url that displays the results.

3. To view results, we open up a web browser and go to the url mentioned in the command prompt. An example would be: http://localhost:7007/#scalars. A window as shown in figure 18 should open up in the browser window.

The Tensorboard window will display the various losses at each step during the training. The AdamOptimizer loss is the total loss computed by the network. The "L2_continuity", "L2_momentum_x" and "L2_momentum_y" determine the L2 loss computed for the continuity and Navier Stokes equations in x and y direction respectively. The "L2_u" and "L2_v" determine how well the boundary conditions are satisfied (soft constraints).

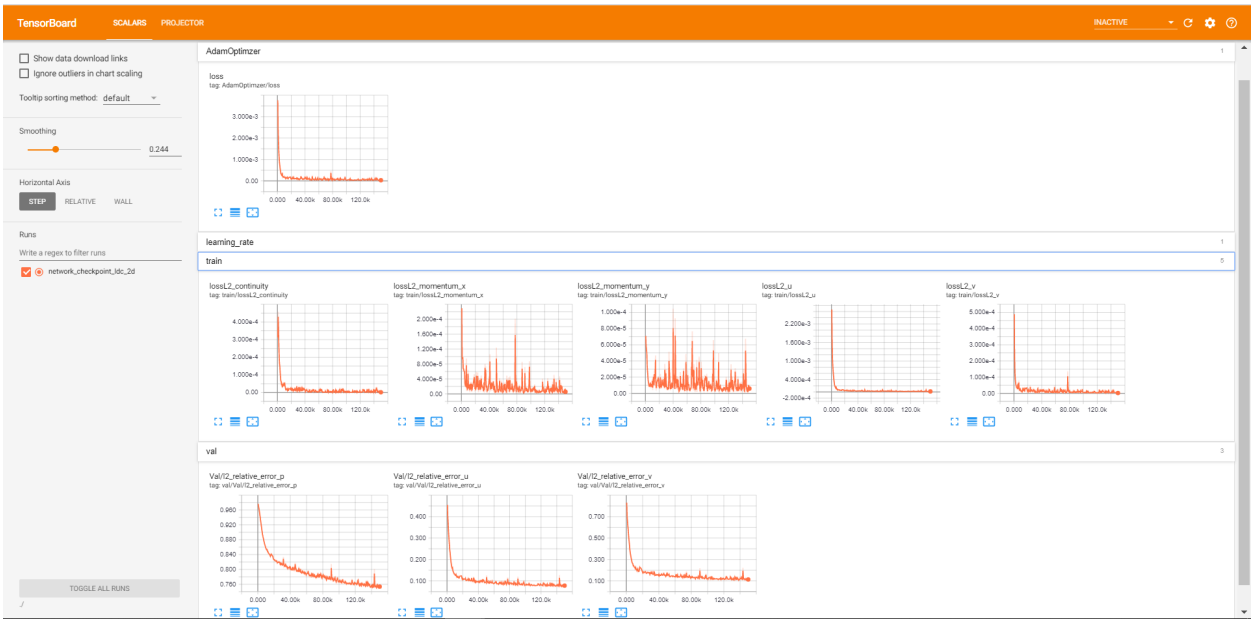


Figure 18: Tensorboard Interface.

2.5.2 Trained model

The network directory is saved based on the results recording frequency specified in the `'rec_results_freq'`. The network directory folder (in this case `'./network_checkpoint_ldc_2d'`) contains the following important files/directories.

1. `checkpoint`: This file specifies the path to the model checkpoint. This is where the information about the latest checkpoint to load is stored. Each checkpoint has a `.index` and `.meta` file. The checkpoint is named as `model.ckpt-XXXX`, where `XXXX` determines the steps count at which the file was written. Depending on how long the simulation was run and the record frequency, the number of checkpoint files in the directory may vary.
2. `train_domain`: This directory contains the data computed on the points defined in the `LDCTrain` class. The data is present in the form of .vtu as well as .npz files. The .vtu files can be viewed using visualization tools like Paraview while .npz files can be used to make custom plots. The .vtu/.npz files in this directory will report only those values that were used to define the boundary conditions and equations. For example, the `./train_domain/results/Interior_true.vtu` and `./train_domain/results/Interior_pred.vtu` correspond to the true and the network predicted values for variables like `'continuity'`, `'momentum_x'` and `'momentum_y'`. Figure 19 shows the comparison between true and computed continuity. This directory is useful to see how well the boundary conditions and equations are being satisfied at the sampled points.
3. `val_domain:` This directory contains the data computed on the points defined in the `LDCVal` class. This domain is more useful in terms of visualizing the true and predicted variables (for this case, flow variables like u, v, p, etc.). The points sampled for predicted data is the same as the points in the true/validation data. The data is again present in the form of .vtu and .npz files. The .vtu files can be viewed using visualization tools like Paraview. The .vtu/.npz files in this directory will report predicted, true (validation data), and the difference between the two data.

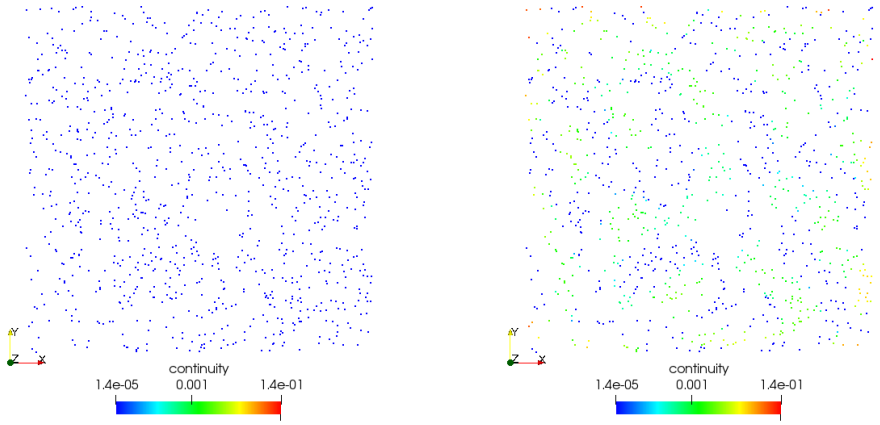


Figure 19: Visualization using Paraview. Left: Continuity as specified in the domain definition. Right: Computed continuity after training.

For example, the `./val_domain/results/Val_true.vtu` and `./train_domain/results/Val_pred.vtu` correspond to the true and the network predicted values for variables like "u", "v" and "p" while `./train_domain/results/Val_diff.vtu` corresponds to the difference between the two. Figure 20 shows the comparison between true and Modulus predicted values of such variables.

Tip: One can use the Delaunay2D feature from Paraview to construct 2D delaunay triangulation from the point cloud (https://kitware.github.io/paraview-docs/latest/python/paraview.simple.Delaunay2D.html)

Tip: One can unpack the .npz files using the use the `numpy.load(filename.npz)` command to get access to all the saved variables. We can then define custom plot functions to generate figures. This was used to create the plot in figure 20.

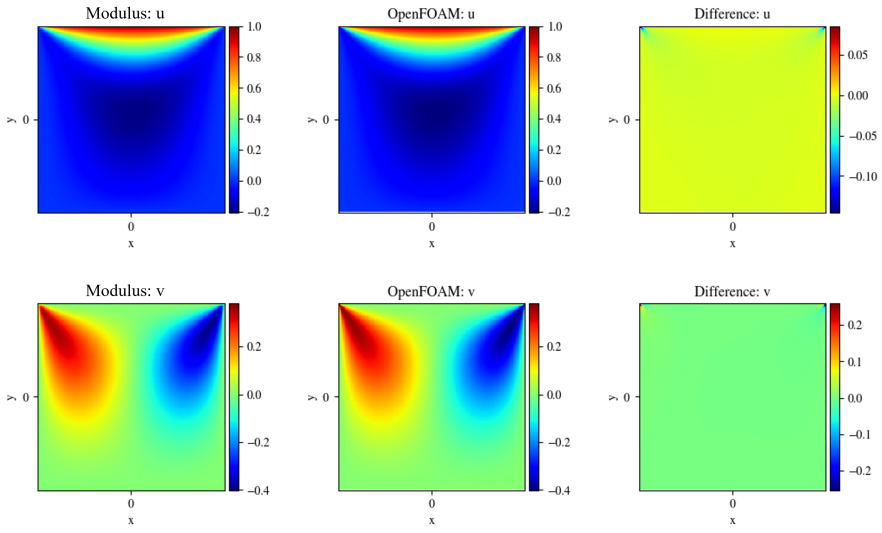


Figure 20: Visualization using custom plot functions. Left: Modulus Prediction, Center: True/Validation Data, Right: Difference

Important: The words `true` and `pred` have different connotations in the `train_domain` and `val_domain`. The below table describes these references more clearly.

	Train domain	Validation domain
true	Expected/Input values specified by you	Analytical/experimental/simulation data
pred	Output values by Modulus after training	Modulus prediction after training

Note: At this point, you might wonder if it is necessary to have Validation data in order to visualize the variables like u, v, and p in the interior. Well, there's no need to worry. We have such cases covered! Modulus is designed to be a stand-alone PDE solver. It presents functionality to create `Inference` and `Monitor` domains to visualize such variables in the absence of validation data. However, we will cover them in tutorial 3. We will also talk more about the use of .npz files to make custom python plots in tutorial 4.

3 Turbulent physics: Zero Equation Turbulence Model

3.1 Introduction

In this tutorial, we will walk through the process of adding a turbulence model to the Modulus simulations. In this tutorial you would learn the following:

1. How to implement the Zero equation turbulence model in Modulus.
2. How to create `Monitor` and `Inference` domains for advanced post-processing.
3. How to create nodes in the graph for arbitrary variables.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of the Modulus user interface.

3.2 Problem Description

In this tutorial we will add the zero equation turbulence for a Lid Driven Cavity flow. The problem setup is similar to the one found in the tutorial 1. The Reynolds number is increased to 1000. The velocity profile is kept the same as the previous problem. To increase the Reynolds Number, the viscosity is reduced to $1 \times 10^{-4}\ m^2/s$.

3.3 Case Setup

The case set up for this tutorial is very similar to the one in tutorial 2. Hence, we will only describe the additions that are made to the previous code.

Note: The python script for this problem can be found at `examples/ldc/ldc_2d_zeroEq.py`.

Importing the required packages

Apart from all the packages imported in tutorial 2, only `ZeroEquation` needs to be imported from the `turbulence_zero_eq` file in the PDES folder.

```
from sympy import Symbol, Eq, Abs
import tensorflow as tf

from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain, MonitorDomain, InferenceDomain
from modulus.data import Validation, Monitor, Inference
from modulus.sympy_utils.geometry_2d import Rectangle
from modulus.csv_utils.csv_rw import csv_to_dict
from modulus.PDES import NavierStokes, ZeroEquation
from modulus.node import Node
from modulus.controller import ModulusController
```

Listing 7: Importing required packages

3.3.1 Creating Geometry and Defining Boundary conditions and Equations

This section is exactly the same as the tutorial 2 since no boundary conditions or the conservation equations are changed.

```
# params for domain
height = 0.1
width = 0.1
vel = 1.5

# define geometry
rec = Rectangle((-width/2, -height/2), (width/2, height/2))
geo = rec

# define sympy varaibles to parametize domain curves
x, y = Symbol('x'), Symbol('y')

class LDCTrain(TrainDomain):
  def __init__(self, **config):
```

```
    super(LDCTrain, self).__init__()
    # top wall
    topWall = geo.boundary_bc(outvar_sympy={'u': vel, 'v': 0},
                              batch_size_per_area=10000,
                              lambda_sympy={'lambda_u': 1.0 - 20*Abs(x), # weight edges to be zero
                                            'lambda_v': 1.0},
                              criteria=Eq(y, height/2))
    self.add(topWall, name="TopWall")

    # no slip
    bottomWall = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                                 batch_size_per_area=10000,
                                 criteria=y < height/2)
    self.add(bottomWall, name="NoSlip")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0},
                               bounds={x: (-width/2, width/2), y: (-height/2, height/2)},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                                             'lambda_momentum_x': geo.sdf,
                                             'lambda_momentum_y': geo.sdf},
                               batch_size_per_area=400000)
    self.add(interior, name="Interior")
```

Listing 8: Defining geometry, boundary conditions and the equations

3.3.2 Creating Validation data

Here, we will use data from an OpenFOAM simulation of LDC flow using Zero Equation turbulence model for validation.

```
# validation data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'U:0': 'u', 'U:1': 'v', 'p': 'p', 'nuT': 'nu'}
openfoam_var = csv_to_dict('openfoam/cavity_uniformVel_zeroEqn_refined.csv', mapping)
openfoam_invar_numpy = {key: value for key, value in openfoam_var.items() if key in ['x', 'y']}
openfoam_outvar_numpy = {key: value for key, value in openfoam_var.items() if key in ['u', 'v', 'nu']}
openfoam_outvar_numpy['nu'] += 1.0e-4

class LDCVal(ValidationDomain):
  def __init__(self, **config):
    super(LDCVal, self).__init__()
    # valication data from openfoam
    val = Validation.from_numpy(openfoam_invar_numpy, openfoam_outvar_numpy)
    self.add(val, name='Val')
```

Listing 9: Defining the Validation data

3.3.3 Creating Monitor and Inference domain

3.3.3.1 Monitor Modulus allows you to monitor desired quantities by plotting them every fixed number of iterations in Tensorboard as the simulation progresses, and analyze the the convergence based on the relative changes in the monitored quantities. A `MonitorDomain` can be used to create such an feature. Examples of such quantities can be point values of variables, surface averages, volume averages or any derived quantities that can be formed using the variables being solved.

The flow variables are available as TensorFlow tensors. You can perform tensor operations available in TensorFlow to create any desired derived variable of your choice. In the code below, we create monitors for continuity and momentum imbalance in the interior.

The points to sample can be selected in a similar way as we did for specifying the `LDCTrain` domain. The `LDCMonitor` class can be created by inheriting from the `MonitorDoamin` parent class.

Note: Apart from the flow variables being solved, you have access to variables like `'normal_x'`, `'normal_y'`, `'area'` and `'sdf'` which are created when we generate the geometry.

```
class LDCMonitor(MonitorDomain):
  def __init__(self, **config):
    super(LDCMonitor, self).__init__()
    # metric for mass imbalance, momentum imbalance and peak velocity magnitude
    global_monitor = Monitor(geo.sample_interior(400000, bounds={x: (-width/2, width/2), y: (-height/2, height
     /2)}),
```

```
                                {'mass_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['continuity']))
    ,
                                 'momentum_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
    momentum_x'])+tf.abs(var['momentum_x'])),
                                })
    self.add(global_monitor, 'GlobalMonitor')
```

Listing 10: Defining the Monitors

3.3.3.2 Inference As suggested in the earlier tutorial, it is not mandatory to have validation data to view the flow variables in the interior. We will now see how to create an Inference domain to plot the desired variables in the interior at a desired point density.

The `LDCInference` class can be created by inheriting from the `InferenceDomain` parent class. The points are again sampled in a similar way as done during the definition of `LDCTrain` and `LDCMonitor` domains. Here, we specify the variables to output as u, v, p and the effective viscosity nu ($\nu + \nu_t$).

```
class LDCInference(InferenceDomain):
  def __init__(self, **config):
    super(LDCInference, self).__init__()
    # save entire domain
    interior = Inference(geo.sample_interior(1e06, bounds={x: (-width/2, width/2), y: (-height/2, height/2)}),
                         ['u','v','p','nu'])
    self.add(interior, name="Inference")
```

Listing 11: Defining Inference domain

Note: During execution, the validation, inference and monitor domains are only computed on the iterations when the results are recorded. By default the results are recorded every 1000 steps (can be controlled using `--rec_results_freq` flag). Thus the density of the points in these domains does not affect the time/iteration as it is not part of the training. However, care should be taken while choosing the batch size, as it can still slow the result recording steps.

Summary on the different domains in Modulus

Between this and the previous tutorials, we have covered all the available domain classes in Modulus. Let's summarize the function and use of each of these:

- `TrainDomain`: Used to define the boundary conditions, initial conditions and the points to minimize the equation residuals while training.
- `ValidationDomain`: Used to compare Modulus solutions with any analytical/simulation/available data.
- `MonitorDomain`: Used to make monitors to check convergence based on desired quantities in Tensorboard.
- `InferenceDomain`: Used to plot data on arbitrary points in the geometry. Useful to visualize the distribution of variables over the entire or subset of problem domain (e.g. portion of interior, only specific walls, etc.).

3.3.4 Adding Turbulence Equation and Making the Neural Network solver

The solver is defined by inheriting the `Solver` parent class as did in the previous tutorials. This time, along with `train_domain` and `val_domain`, `inference_domain` and `monitor_domain` are are assigned.

The equations to be solved are specified under `self.equations`. In addition to the Navier Stokes equation, the Zero Equation turbulence model is included by instantiating the `ZeroEquation` equation class. The kinematic viscosity ν is now a string variable in the Navier Stokes equation. The Zero equation turbulence model supplies the effective viscosity $(\nu + \nu_t)$ to the Navier Stokes equations. The kinematic viscosity of the fluid calculated based on the Reynolds number is supplied as an input to the `ZeroEquation` class.

The Zero Equation turbulence model is defined in the Equations 60, 61, 62 Note $\mu_t = \rho\nu_t$.

$$\mu_t = \rho l_m^2 \sqrt{G} \tag{60}$$

$$G = 2(u_x)^2 + 2(v_y)^2 + 2(w_z)^2 + (u_y + v_x)^2 + (u_z + w_x)^2 + (v_z + w_y)^2 \tag{61}$$

$$l_m = \min(0.419d, 0.09d_{max}) \tag{62}$$

Where, l_m is the mixing length, d is the normal distance from wall, d_{max} is maximum normal distance and $\sqrt{G}$ is the modulus of mean rate of strain tensor.

Please take a look into the source code documentation of turbulence model for more details.

As we see, the zero equation turbulence model requires normal distance from no slip walls to compute the turbulent viscosity. This information is supplied in the form of variable `'normal_distance'` by making a node from the SDF function. `node_from_sympy` can be used to create such nodes for arbitrary variables.

The inputs and the outputs of the neural network are specified and the nodes of the architecture are made as before.

```
class LDCSolver(Solver):
  train_domain = LDCTrain
  val_domain = LDCVal
  monitor_domain = LDCMonitor
  inference_domain = LDCInference

  def __init__(self, **config):
    super(LDCSolver, self).__init__(**config)
    self.equations = (NavierStokes(nu='nu', rho=1.0, dim=2, time=False).make_node()
                      + ZeroEquation(nu=1.0e-4, dim=2, time=False, max_distance=0.05).make_node()
                      + [Node.from_sympy(geo.sdf, 'normal_distance')])
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_ldc_2d_zeroEq',
        'start_lr': 3e-4,
        'decay_steps': 20000,
        'max_steps': 1000000
        })

if __name__ == '__main__':
  ctr = ModulusController(LDCSolver)
  ctr.run()
```

Listing 12: Defining the Neural Network Solver

3.4 Running the Modulus solver

Once the python file is set up, one can save the file and exit out of the text editor. In the command prompt, type in:

```
python ldc_2d_zeroEq.py
```

3.5 Results and Post-processing

3.5.1 Setting up Tensorboard

The Tensorboard can be set up as shown in tutorial 2. This time, apart from the AdamOptimizer, learning_rate, train, and val tabs, you will see an additional monitor tab. This tab will plot the quantities that you chose to monitor, during each iteration (figure 21).

3.5.2 Trained model

In addition to the `checkpoint` file, `train_domain`, and `val_domain` directories, you can notice two additional directories for `monitor_domain` and `inference_domain` created in the network directory.

1. `monitor_domain`: This directory contains the information required for plotting the monitors. The `./monitor_domain/results/` contains a file named `GlobalMonitor.csv` which you can be use to export the monitors data to an external plotting tool like gnuplot. An example of the plot generated by gnuplot and also be found in the same directory (`GlobalMonitor.png`).
2. `inference_domain`: This directory contains the data computed on the points defined in the `LDCFInference` class. The data is present in the form of .vtu files. The file `./results/Inference.vtu` can be viewed using Paraview to view all the selected variables on the chosen points during the definition of `LDCInference`.

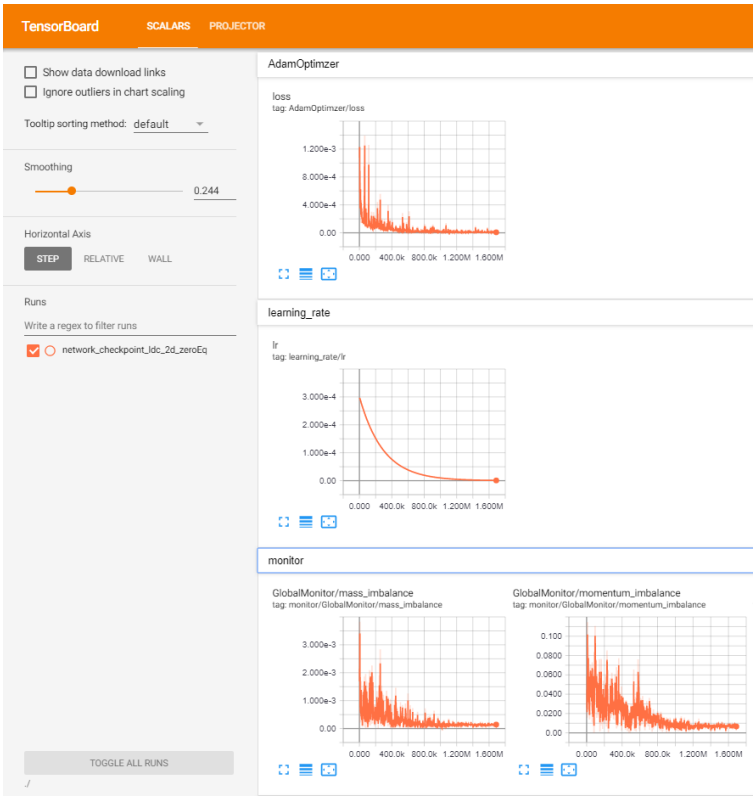


Figure 21: Visualizing Monitors in Tensorboard

Figure 22 shows the effective viscosity nu $(\nu + \nu_t)$ outputted by the Zero Equation turbulence model and plotted using the Inference domain.

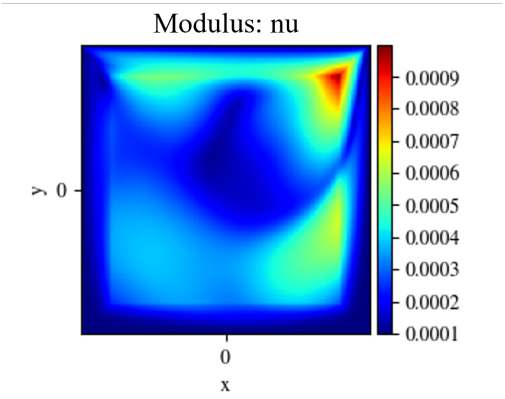


Figure 22: Visualizing variables from Inference domain

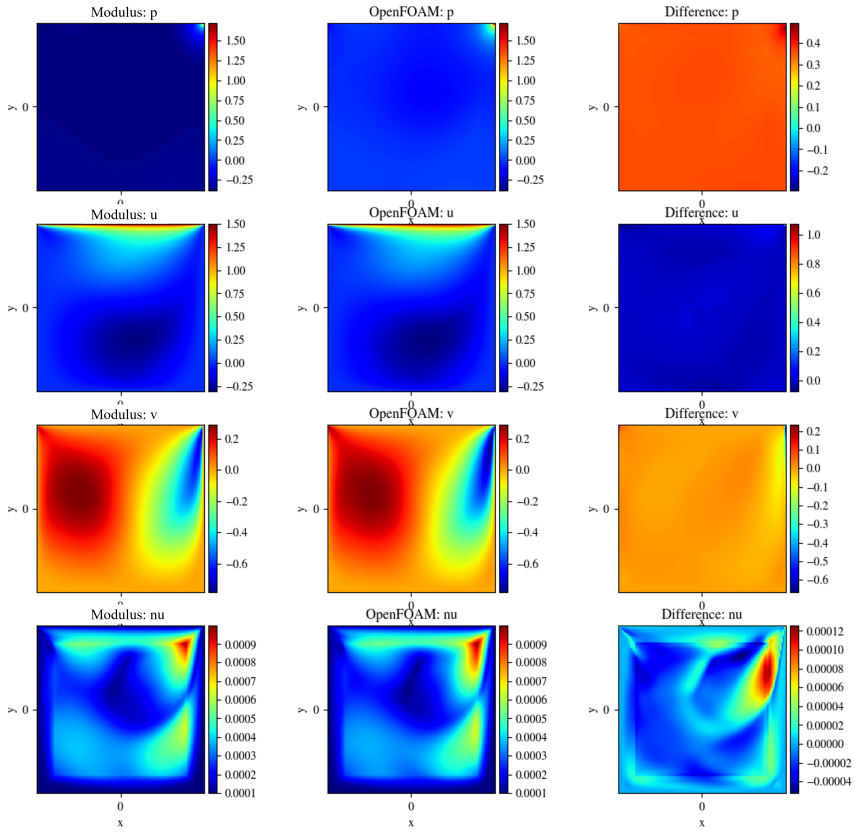


Figure 23: Comparison with OpenFOAM data. Left: Modulus Prediction. Center: OpenFOAM, Right: Difference

4 Transient physics: Wave Equation

4.1 Introduction

In this tutorial, we will walk through the process of setting up a custom PDE in Modulus. We will demonstrate the process on a time-dependent problem of a simple 1D wave equation. This way we will also show how to solve transient physics in Modulus. In this tutorial you would learn the following:

1. How to write your own Partial Differential Equation and boundary conditions in Modulus.
2. How to solve a time-dependent problem in Modulus.
3. How to impose time boundary conditions.
4. How to generate validation data from analytical solutions.

Prerequisites

This tutorial assumes that you have completed the tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of Modulus user interface.

4.2 Problem Description

In this tutorial, we will solve a simple 1D wave equation . The wave is described by the below equation.

$$\begin{aligned} u_{tt} &= c^2 u_{xx} \\ u(0,t) &= 0, \\ u(\pi,t) &= 0, \\ u(x,0) &= \sin(x), \\ u_t(x,0) &= \sin(x). \end{aligned} \tag{63}$$

Where, the wave speed $c = 1$ and the analytical solution to the above problem is given by $\sin(x)(\sin(t) + \cos(t))$.

4.3 Writing custom PDEs and boundary/initial conditions

Introduction

`custom PDE` For this tutorial, we will write the 1D wave equation. The wave equation in n-dimensions can be referred at https://en.wikipedia.org/wiki/Wave_equation. We will also see how to handle derivative type boundary conditions. The PDEs defined in the source code directory `modulus/PDES/` can be used for reference.

We will make a file `wave_equation.py` and define the wave equation in 1D in it. The `PDES` class allows you to write the equations symbolically in Sympy. This allows users to quickly write their equations in the most natural way possible. The Sympy equations are converted to TensorFlow expressions in the back-end and can also be printed to ensure correct implementation.

Implementing Equation

First we will create a class `WaveEquation1D` that inherits from `PDES`.

```
from sympy import Symbol, Function, Number

from modulus.pdes import PDES
from modulus.variables import Variables

class WaveEquation1D(PDES):
  name = 'WaveEquation1D"
```

Listing 13: Coding your own PDE: Part 1

Now we will create the initialization method for this class that defines the equation(s) of interest. We will be defining the wave equation using the wave speed(c). If c is given as a string we will convert it to functional form. This will allow us to solve problems with spatially/temporally varying wave speed. This will also be used in the subsequent inverse example.

```
  def __init__(self, c=1.0):
    # coordinates
    x = Symbol('x')

    # time
    t = Symbol('t')

    # make input variables
    input_variables = {'x':x,'t':t}

    # make u function
    u = Function('u')(*input_variables)

    # wave speed coefficient
    if type(c) is str:
      c = Function(c)(*input_variables)
    elif type(c) in [float, int]:
      c = Number(c)

    # set equations
    self.equations = Variables()
    self.equations['wave_equation'] = u.diff(t, 2) - (c**2*u.diff(x)).diff(x)
```

Listing 14: Coding your own PDE: Part 2

First we defined the input variables x and t with Sympy symbols. Then we defined the functions for u and c that are dependent on x and t. Using these we can write out our simple equation $u_{tt} = (c^2 u_x)_x$. We store this equation in the class by adding it to the dictionary of equations.

Note the structure of the equation for `'wave_equation'`. We will have to move all the terms of the PDE either to LHS or RHS and just have the source term on one side. This way, while using the equations in the TrainDomain, we can assign a custom source function to the `'wave_equation'` key instead of 0 to add the source to our PDE.

Great! We just wrote our own PDE in Modulus! To verify the implementation, you can refer to the file `modulus/PDES/wave_equation.py`. Also, once you have understood the process to code a simple PDE, you can easily extend the procedure for different PDEs in multi-dimensions (2d, 3d, etc.) by making additional input variables, constants, etc. You can also bundle multiple PDEs together in a same file by adding new keys to the equations dictionary.

Now let's head to writing the solver file where we make use of the newly coded wave equation to solve the 1D wave problem.

4.4 Case Setup

In this tutorial, we will make use of `Line1D` to sample points in a single dimension. The time-dependent equation is solved by supplying t as a variable parameter to the `param_ranges` , with the ranges being the time domain of interest. `param_ranges` is also used when solving problems involving variation in geometric or variable PDE constants.

Note: We solve the problem by treating time as a continuous variable . Discrete time stepping is still under development. At the end of the tutorial we will present results from some of the architectures that are currently under development.

Note: The python script for this problem can be found at `examples/wave_equation/wave_1d.py`. We have also provided the PDE we coded in `wave_equation.py` also in the same directory for reference.

Importing the required packages

The new packages/modules imported in this tutorial are `geometry_1d` for using the 1D geometry. We will source the `WaveEquation1D` from the file we just created.

```
from sympy import Symbol, sin
import numpy as np

from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain
from modulus.data import Validation
from modulus.sympy_utils.geometry_1d import Line1D
from modulus.controller import ModulusController

from wave_equation import WaveEquation1D
```

Listing 15: Importing the required packages and modules

4.4.1 Creating Geometry and Defining Initial and Boundary conditions and Equations to solve

For generating geometry of this problem, we will use the `Line1D(pt1, pt2)`. The boundaries for `Line1D` are the end points and the interior covers all the points in between the two endpoints.

As described earlier, we will use the `param_ranges` attribute to solve for time. For defining the initial conditions, we will set `param_ranges={t_symbol: 0.0}`. We will solve the wave equation for $t = (0, 2\pi)$. The derivative boundary condition can be handled by specifying the key `'u__t'`. The derivatives of the variables can be specified by adding `'__t'` for time derivative and `'__x'` for spatial derivative (`'u__x'` for $\partial u/\partial x$, `'u__x__x'` for $\partial^2 u/\partial x^2$, etc.).

The below code uses these tools to generate the geometry, boundary conditions and the equations.

```
# params for domain
L = float(np.pi)

# define geometry
geo = Line1D(0, L)

# define sympy varaibles to parametize domain curves
t_symbol = Symbol('t')
time_range = {t_symbol: (0, 2*L)}

class WaveTrain(TrainDomain):
  def __init__(self, **config):
    super(WaveTrain, self).__init__()
    # sympy variables
    x = Symbol('x')

    #initial conditions
    IC = geo.interior_bc(outvar_sympy={'u': sin(x), 'u__t': sin(x)},
                         bounds={x: (0, L)},
                         batch_size_per_area=100,
                         lambda_sympy={'lambda_u': 1.0,
                                       'lambda_u__t': 1.0},
                         param_ranges={t_symbol: 0.0})
    self.add(IC, name="IC")

    #boundary conditions
    BC = geo.boundary_bc(outvar_sympy={'u': 0},
                         batch_size_per_area=100,
                         lambda_sympy={'lambda_u': 1.0},
                         param_ranges=time_range)
    self.add(BC, name="BC")

    # interior
    interior = geo.interior_bc(outvar_sympy={'wave_equation': 0},
                               bounds={x: (0, L)},
                               batch_size_per_area=1000,
                               lambda_sympy={'lambda_wave_equation': 1.0},
                               param_ranges=time_range)
    self.add(interior, name="Interior")
```

Listing 16: Defining the geometry, boundary conditions and equations

4.4.2 Creating in Validation data from analytical solutions

For this problem, the analytical solution is easy and can be solved simultaneously instead of importing a .csv file. The below code shows the process define such a dataset.

```
class WaveVal(ValidationDomain):
  def __init__(self, **config):
    super(WaveVal, self).__init__()
    # make validation data
    deltaT = 0.01
    deltaX = 0.01
    x = np.arange(0, L, deltaX)
    t = np.arange(0, 2*L, deltaT)
    X, T = np.meshgrid(x, t)
    X = np.expand_dims(X.flatten(), axis=-1)
    T = np.expand_dims(T.flatten(), axis=-1)
    u = np.sin(X) * (np.cos(T) + np.sin(T))
    invar_numpy = {'x': X, 't': T}
    outvar_numpy = {'u': u}
    val = Validation.from_numpy(invar_numpy, outvar_numpy)
```

```
    self.add(val, name='Val')
```

Listing 17: Generating validation data

4.4.3 Making the Neural Network Solver

This part of of the problem is similar to the tutorial 2. Equation `WaveEquation` is used to compute the wave equation and the wave speed is defined based on the problem statement.

```
# Define neural network
class WaveSolver(Solver):
  train_domain = WaveTrain
  val_domain = WaveVal

  def __init__(self, **config):
    super(WaveSolver, self).__init__(**config)

    self.equations = WaveEquation1D(c=1.0).make_node()
    wave_net = self.arch.make_node(name='wave_net',
                                   inputs=['x', 't'],
                                   outputs=['u'])
    self.nets = [wave_net]

  @classmethod # Explain This
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_wave',
        'max_steps': 10000,
        'decay_steps': 100,
        })

if __name__ == '__main__':
  ctr = ModulusController(WaveSolver)
  ctr.run()
```

Listing 18: Defining the Neural Network Solver

4.5 Running the Modulus solver

Once the python file is setup, you can save the file and exit out of the text editor. In the command prompt, type in:

```
python wave_1d.py
```

4.6 Results and Post-processing of non-standard datasets

Since this is a 1D, time dependent problem, it is difficult to visualize the .vtu files. As seen in tutorial 2, the network directory `./validation_domain/results/` also saves the data in the form of .npz arrays which can be unpacked using the python command `numpy.load(filename.npz)` to get access to all the saved variables. We can then define custom plot functions and generate figures similar to one shown in figure 24.

Note: One can still load the .vtu files and visualize by defining new x, y, z coordinates using the Calculator feature from Paraview. However, we won't cover this application in this tutorial.

4.7 Temporal loss weighting and time-marching schedule

We have observed that two simple tricks, namely temporal loss weighting and time-marching schedule, can improve the performance of the continuous time approach for transient simulations. The idea behind the temporal loss weighting is to weight the loss terms temporally such that the terms corresponding to earlier times have a larger weight compared to those corresponding to later times in the time domain. For example, our temporal loss weighting can take the following linear form:

$$\lambda_T = C_T \left(1 - \frac{t}{T}\right) + 1. \tag{64}$$

Here, λ_T is the temporal loss weight, C_T is a constant that controls the weight scale, T is the upper bound for the time domain, and t is time.

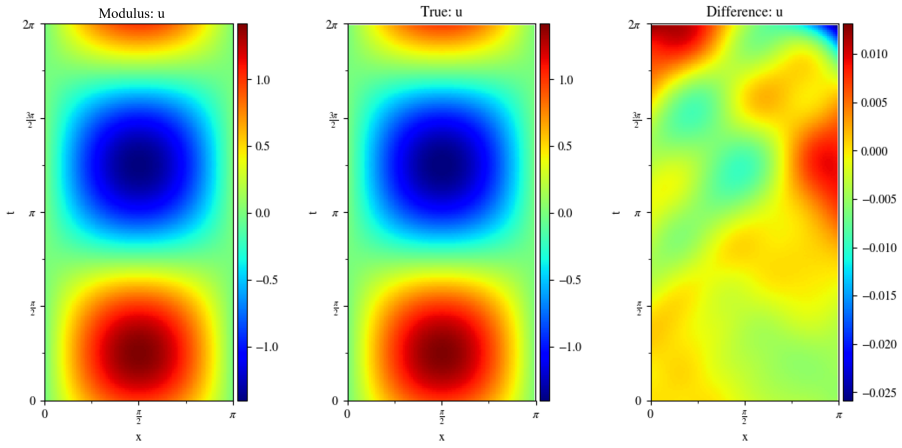


Figure 24: Left: Modulus. Center: Analytical Solution. Right: Difference

The idea behind the time marching schedule is to consider the time domain upper bound T to be variable and a function of the training iteration s. This variable can then change such that more training iterations are taken for the earlier times compared to later times. Several schedules can be considered, for instance, we can use the following:

$$T_v(s) = \min(1, \frac{2s}{S}), \tag{65}$$

where $T_v(s)$ is the variable time domain upper bound, s is the training iteration number, and S is the maximum number of training iterations. At each training iteration, we will then sample continuously from the time domain in the range of $[0, T_v(s)]$.

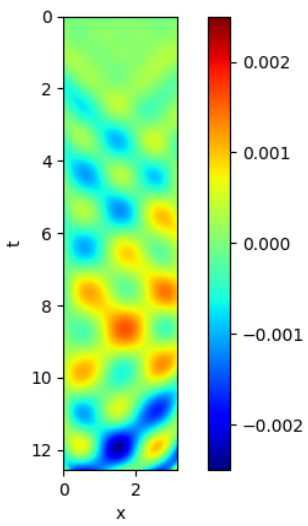
Figures 25, 26, 27 show the Modulus validation error for models trained with and without using temporal loss weighting and time marching for transient 1D and 2D wave examples and a 2D channel flow over a bump. It is evident that these two simple tricks can improve the training accuracy.

4.8 Experimental RNN and GRU architectures for time-domain problems

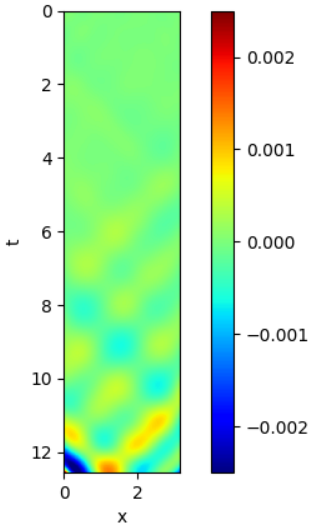
So far we have seen how to solve the time-domain problems using the continuous time approach. In this case, we treat the time as any other continuous variable (eg. Cartesian coordinates). In practice we found that using standard fully connected networks were not suitable for training over long time spans and the accuracy decreases as the time frame of simulation increases.

Currently, we are experimenting with using Recurrent Neural Networks to overcome these deficiencies by modeling the temporal dynamics. This is achieved by temporally discretizing the domain and having the recurrent network make discrete predictions. These discrete predictions are then smoothed using a continuous and differentiable bump function similar to work seen here [27]. These networks have shown to outperform fully connected networks when solving for a longer time duration (Figure 28). This work is still under investigation however we present results using standard Recurrent Neural Networks (RNNs) and Gated Recurrent Units (GRUs) [28] [29]. We hypothesis that the GRUs outperform the RNNs by alleviating the vanishing gradient problem.

These experimental architectures can be found in `examples/wave_equation/` directory.

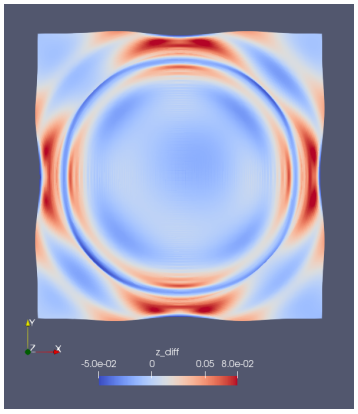


(a) Continuous time without temporal weighting and time marching

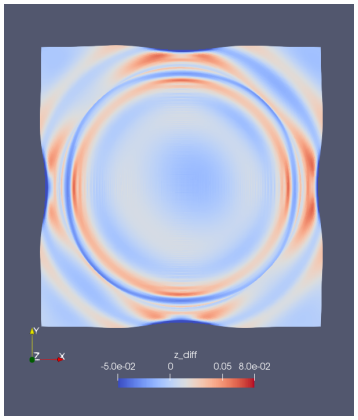


(b) Continuous time with temporal weighting and time marching

Figure 25: Modulus validation error for the 1D transient wave example: (a) standard continuous time approach; (b) continuous time approach with temporal loss weighting and time marching.



(a) Continuous time without temporal weighting and time marching



(b) Continuous time with temporal weighting and time marching

Figure 26: Modulus validation error for the 2D transient wave example: (a) standard continuous time approach; (b) continuous time approach with temporal loss weighting and time marching.

(a) Continuous time without temporal weighting and time marching

(b) Continuous time with temporal weighting

Figure 27: Modulus validation error for a 2D transient channel flow over a bump: (a) standard continuous time approach; (b) continuous time approach with temporal loss weighting.

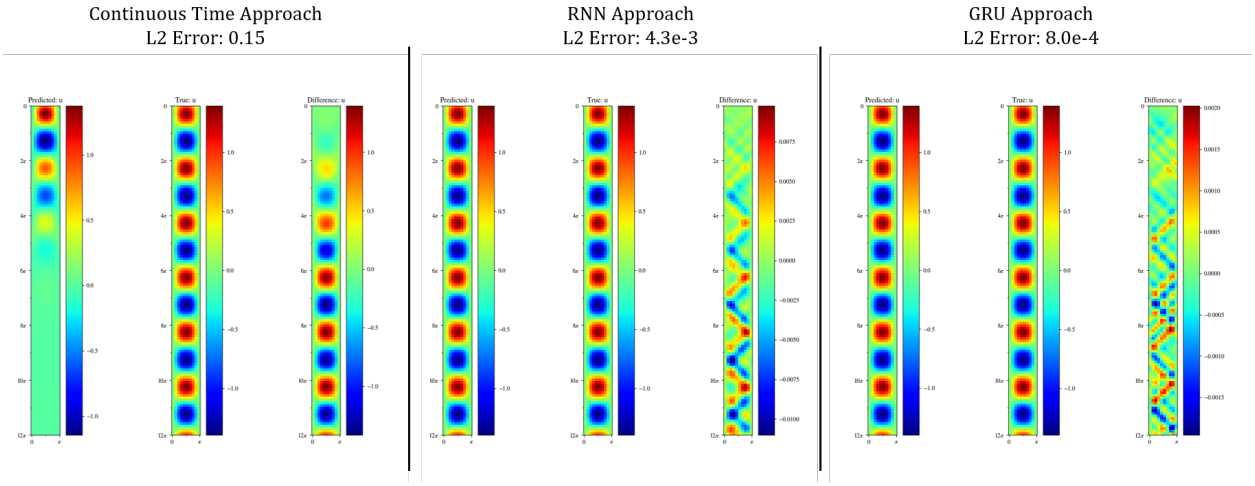


Figure 28: Comparisons of the new experimental architectures vs. continuous time approach

5 Transient Physics: 2D Seismic Wave Propagation

5.1 Introduction

In this tutorial, we extend the previous 1D wave equation example and solve a 2D seismic wave propagation problem commonly used in seismic survey. In this tutorial, you would learn the following:

1. How to solve a 2d time-dependant problem in Modulus
2. How to define open boundary condition as custom equations
3. How to present variable velocity model as an additional network

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity flow and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you refer to tutorial 4 for information on defining new differential equations, and solving time-dependent problems in Modulus.

Seismic survey modelling workflow

In this example we aim to use Modulus to solve the 2D acoustic wave equation with a single Ricker source in a layered velocity model. The core processes we are aiming to model is a seismic survey which consists of two main components.

- **Source:** A source is positioned at a single or a few physical locations where artificial pressure is injected into the domain we want to model. In case of land survey, it is usually dynamite blowing up at a given location, or a vibroseis (a vibrating engine generating continuous sound waves). For a marine survey, the source is an air gun sending a bubble of compressed air into the water that will expand and generate a seismic wave.
- **Receiver:** A set of microphones or hydrophones are used to measure the resulting wave and create a set of measurements called a Shot Record. These measurements are recorded at multiple locations.

5.2 Problem Description

The acoustic wave equation for the square slowness, defined as $m = 1/c^2$ where c is the speed of sound (velocity) of a given physical medium with constant density, and a source q is given by:

$$u_{tt} = c^2 u_{xx} + c^2 u_{yy} + q \quad \text{in } \Omega \tag{66}$$

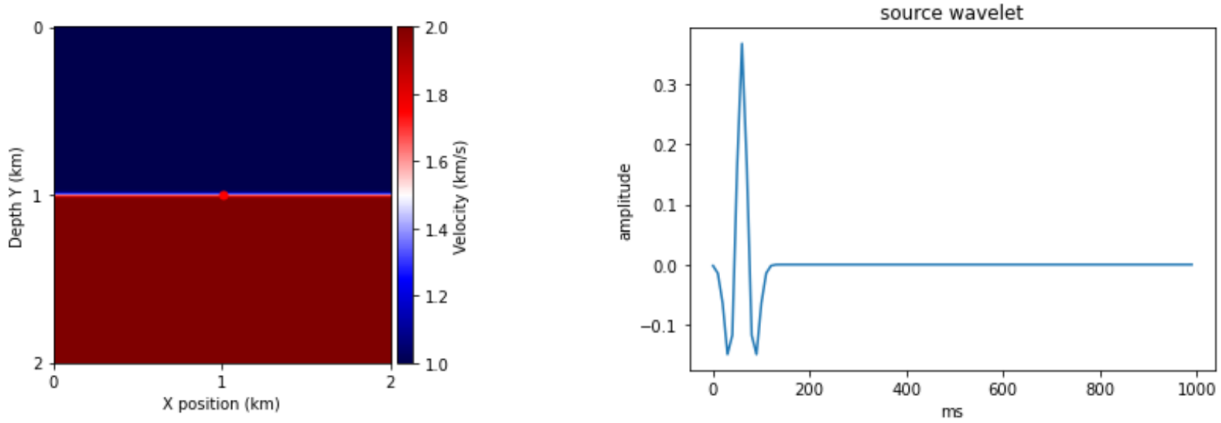
Where $u(\mathbf{x}, t)$ represents the pressure response (known as "wavefield") at location vector $\mathbf{x}$ and time t in an acoustic medium. The problem is solved with zero initial conditions to guarantee unicity of the solution Despite its linearity, the wave equation is notoriously challenging to solve in complex media, because the dynamics of wavefield at the interfaces of the media can be highly complex, with multiple types of waves with large range of amplitudes and frequencies interfering simultaneously.

Source in seismic survey is positioned at a single or a few physical locations as described above. In this tutorial, we solve the 2D acoustic wave equation with a single Ricker Source in a layered velocity model, 1.0 km/s at the top layer and 2.0 km/s the bottom (Figure 29 left).

In our problem, we set domain size 2 km x 2 km, and a single source is located at the center of the domain. The source term signature is modelled using a Ricker wavelet, given by Equation 67 and illustrated in Figure 29 right, with a peak wavelet frequency $f_0 = 15Hz$

$$q(t) = \left(1 - 2\pi^2 f_0^2 (t - \frac{1}{f_0})^2\right) \, e^{-\pi^2 f_0^2 \left(t - \frac{1}{f_0}\right)} \tag{67}$$

We use wavefield data at earlier time steps (150 ms - 300 ms) generated from finite difference simulations, using Devito [30, 31] (https://github.com/devitocodes/devito/tree/master/examples/seismic/tutorials), as constraints of temporal boundary conditions, and train Modulus to produce wavefields at later time steps (300 ms – 1000 ms).



(a) Velocity model of the medium

(b) Ricker source signature

Figure 29: Seismic domain (left) with a single Ricker wavelet source signature (right) located at the center.

5.3 Case Setup

The case setup for this problem is similar to tutorial 4. Hence, we discuss only the main highlights of this problem.

Note: The python script for this problem can be found at `examples/seismic_wave/`.

5.3.1 Defining the Equations

A second-order PDE requires strict BC on both the initial wavefield and its derivatives for its solution to be unique. In the field, the seismic wave propagates in every direction to an "infinite" distance. In the finite computational domain, Absorbing BC (ABC) or Perfectly Matched Layers (PML) are artificial boundary conditions typically applied in conventional numerical approaches to approximate an infinite media by damping and absorbing the waves at the limit of the domain, in order to avoid reflections from the boundary. However, NN solver is meshless – it is not suitable to implement ABC or PML. To enable a wave to leave the computational domain and travel undisturbed through the boundaries, we apply an open boundary condition, also called a radiation condition, which imposes the first-order PDEs at the boundaries. More information on the impact of boundary conditions in a wave equation can be found at http://hplgit.github.io/wavebc/doc/pub/._wavebc_cyborg002.html.

$$
\begin{aligned}
&\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0, \quad \text{at } x = 0 \\
&\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \quad \text{at } x = dLen \\
&\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial y} = 0, \quad \text{at } y = 0 \\
&\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial y} = 0, \quad \text{at } y = dLen
\end{aligned}
\tag{68}
$$

Previous tutorials have described how to define custom PDEs. Similarly we create a class `OpenBoundary` that inherits from `PDES`.

```
# define open boundary conditions
class OpenBoundary(PDES):

  name = 'OpenBoundary'

  def __init__(self, u='u', c='c', dim=3, time=True):
    # set params
    self.u = u
    self.dim = dim
    self.time = time
```

```
    # coordinates
    x, y, z = Symbol('x'), Symbol('y'), Symbol('z')

    # normal
    normal_x, normal_y, normal_z = Symbol('normal_x'), Symbol('normal_y'), Symbol('normal_z')

    # time
    t = Symbol('t')

    # make input variables
    input_variables = {'x':x,'y':y,'z':z,'t':t}
    if self.dim == 1:
      input_variables.pop('y')
      input_variables.pop('z')
    elif self.dim == 2:
      input_variables.pop('z')
    if not self.time:
      input_variables.pop('t')

    # Scalar function
    assert type(u) == str, "u needs to be string"
    u = Function(u)(*input_variables)

    # wave speed coefficient
    if type(c) is str:
      c = Function(c)(*input_variables)
    elif type(c) in [float, int]:
      c = Number(c)

    # set equations
    self.equations = {}
    self.equations['open_boundary'] = (u.diff(t)
                                       + normal_x * c*u.diff(x)
                                       + normal_y * c*u.diff(y)
                                       + normal_z * c*u.diff(z))
```

Listing 19: Defining custom PDEs to represent the open boundary condition

5.3.2 Variable Velocity Model

In tutorial 4, velocity (c) in the physical medium is a constant. In this problem, we have velocity model that varies with locations x. We use a $tanh$ function form to represent velocity c as a function of x and y:

Note: A $tanh$ function was used to avoid the sharp discontinuity at the interface. Such smoothing of sharp boundaries helps the convergence of the Neural network solver.

```
# define velocity model 2.0 km/s at the bottom and 1.0 km/s at the top using tanh function
mesh_x, mesh_y = np.meshgrid(np.linspace(0, 2, 512),
                             np.linspace(0, 2, 512),
                             indexing='ij')
wave_speed_invar = {}
wave_speed_invar['x'] = np.expand_dims(mesh_x.flatten(), axis=-1)
wave_speed_invar['y'] = np.expand_dims(mesh_y.flatten(), axis=-1)
wave_speed_outvar = {}
wave_speed_outvar['c'] = np.tanh(80*(wave_speed_invar['y']-1.0))/2 + 1.5
```

Listing 20: Defining variable velocity model

5.3.3 Solving the PDEs: Creating geometry, defining training and validation domains, making the Neural Network solver

Now that we have PDEs defined, we need to first create the 2D geometry of computational domain, symbols for x, y, and time t, as well as time range of the solutions.

```
# define geometry
dLen = 2 #km
geo = Rectangle((0, 0),
                (dLen, dLen))

# define sympy variables to parametrize domain curves. 'u' is the wavefield 'u'
x, y = Symbol('x'), Symbol('y')

# define time domain, from 0 to 1000 ms
t_symbol = Symbol('t')
```

```
time_length = 1
time_range = {t_symbol: (0.15, time_length)}
```

Listing 21: Generating Geometry

Next, we will define the training domain. Note that we added `wave_speed`, i.e., the variable velocity we defined earlier. We also read in four training datasets, i.e., the wavefield data generated at earlier time steps using Devito, to constrain the NN as boundary conditions. For time-dependent problems, time t is considered as a special component of $\mathbf{x}$.

```
class WaveTrain(TrainDomain):
  def __init__(self, **config):
    super(WaveTrain, self).__init__()
    wave_speed = BC.from_numpy(wave_speed_invar, wave_speed_outvar, batch_size=2048//2)
    self.add(wave_speed, "WaveSpeed")

    batch_size = 1024
    #for i, ms in enumerate(np.linspace(150, 225, 4)):
    for i, ms in enumerate(np.linspace(150,300,4)):
      val_invar, val_outvar = read_wf_data(ms)
      lambda_weighting = {}
      lambda_weighting['lambda_u'] = np.full_like(val_invar['x'], 10.0/batch_size)
      val = BC.from_numpy(val_invar, val_outvar, batch_size=batch_size, lambda_numpy=lambda_weighting)
      self.add(val, "BC"+str(i).zfill(4))


    # interior
    interior = geo.interior_bc(outvar_sympy={'wave_equation': 0},
                               bounds={x: (0, dLen), y: (0, dLen)},
                               batch_size_per_area=4096,
                               lambda_sympy={'lambda_wave_equation': 0.0001},
                               param_ranges=time_range)
    self.add(interior, name="Interior")

    #boundary conditions
    edges = geo.boundary_bc(outvar_sympy={'open_boundary': 0},
                            batch_size_per_area=1024,
                            lambda_sympy={'lambda_open_boundary': 0.01*time_length},
                            param_ranges=time_range)
    self.add(edges, name="Edges")
```

Listing 22: Defining Train Domain

Validation domain is defined similarly:

```
class WaveVal(ValidationDomain):
  def __init__(self, **config):
    super(WaveVal, self).__init__()

    # make validation data
    #for i, ms in enumerate(np.linspace(250,1000,16)):
    for i, ms in enumerate(np.linspace(350,950,13)):
      val_invar, val_outvar = read_wf_data(ms)
      val = Validation.from_numpy(val_invar, val_outvar)
      self.add(val, "VAL_"+str(i).zfill(4))
```

Listing 23: Defining Validation Domain

Now that we have the definitions for the training data and the validation data complete, we can form the solver and run the problem. Note that both `WaveEquation` and `OpenBoundary` are used for equations. An additional network, with input variables x and y and output c, is added:

```
# Define neural network
class WaveSolver(Solver):
  train_domain = WaveTrain
  val_domain = WaveVal

  def __init__(self, **config):
    super(WaveSolver, self).__init__(**config)

    self.equations = (WaveEquation(u='u', c='c', dim=2, time=True).make_node(stop_gradients=['c'])
                    + OpenBoundary(u='u', c='c', dim=2, time=True).make_node())

    wave_net = self.arch.make_node(name='wave_net',
                                   inputs=['x', 'y', 't'],
```

```
                                    outputs=['u'])
    speed_net = self.arch.make_node(name='speed_net',
                                    inputs=['x', 'y'],
                                    outputs=['c'])

    self.nets = [wave_net, speed_net]


  @classmethod # Explain This
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_seismic_wave',
        'max_steps': 40000,
        'decay_steps': 5000,
        'start_lr': 3e-4,
        'layer_size': 256,
        })

if __name__ == '__main__':
  ctr = ModulusController(WaveSolver)
  ctr.run()
```

Listing 24: Defining equations and making the Neural Network solver

5.4 Results and Post-processing

The results from Modulus simulation are compared against the simulation data generated from Devito. The plots are created using numpy files created in `val_domain` in the network checkpoint. We can see that Modulus results are noticeably better than Devito, predicting wavefield with much less boundary reflection, specially at later time steps.

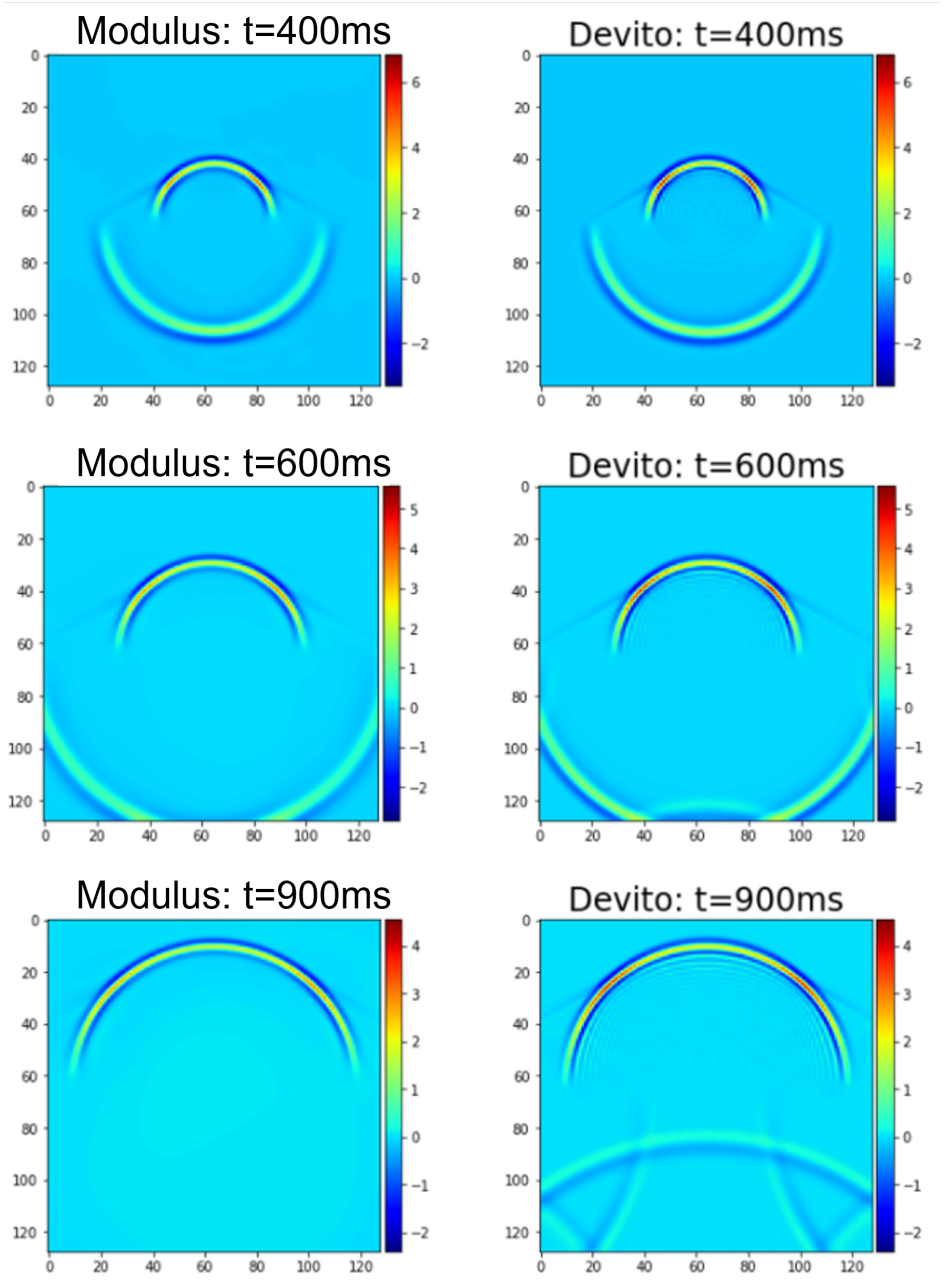


Figure 30: Comparison of Modulus results with Devito solution

6 Transient Navier-Stokes via Moving Time Window: Taylor Green Vortex Decay

6.1 Introduction

In the previous tutorials we have seen solving the transient problems with continuous time approach and also seen a few experimental architectures that can overcome some of the difficulties with continuous time approach. In this tutorial, we present a moving time window approach to solve a complex transient Navier-Stokes problem. In this tutorial, you would learn the following:

1. How to solve sequences of problems/domains in Modulus
2. How to set up periodic boundary conditions

Prerequisites

This tutorial assumes you have completed tutorial 4 on transient simulations.

6.2 Problem Description

As mentioned in tutorial 4, solving transient simulations with only the time stepping method can be difficult for long time duration. In this tutorial, we will show how this can be overcome by using a moving time window. Our example problem is the 3D Taylor-Green vortex decay at Reynolds number 500. The Taylor-Green vortex problem is often used as a benchmark to compare solvers and in this case we will generate validation data with a spectral solver. The domain is a cube of length 2π with periodic boundary conditions on all sides. The initial conditions we will use are,

$$u(x, y, z, 0) = sin(x)cos(y)cos(z) \tag{69}$$
$$v(x, y, z, 0) = -cos(x)sin(y)cos(z) \tag{70}$$
$$w(x, y, z, 0) = 0 \tag{71}$$
$$p(x, y, z, 0) = \frac{1}{16}(cos(2x) + cos(2y))(cos(2z) + 2) \tag{72}$$
$$\tag{73}$$

We will be solving the time dependent incompressible Navier-Stokes equations with a density 1 and viscosity 0.002. We note that because we have periodic boundaries on all sides we do not need boundary conditions 31.

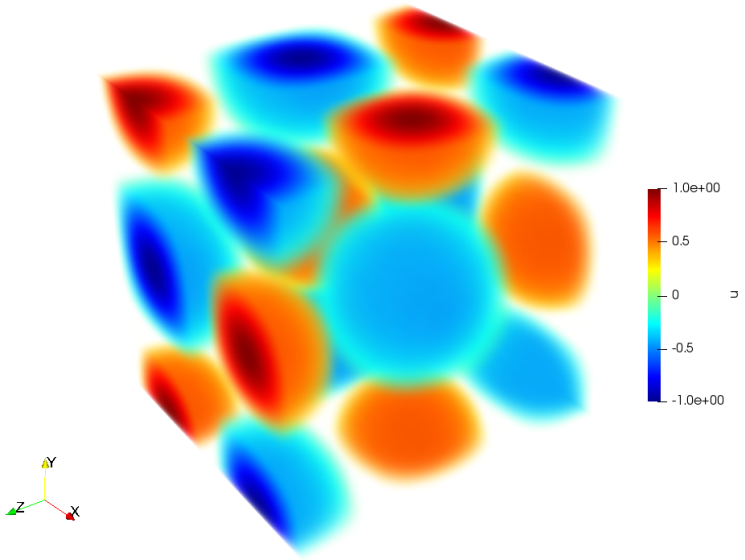


Figure 31: Taylor-Green vortex initial conditions.

The moving time window approach works by iteratively solving for small time windows to progress the simulation forward. The time windows use the previous window as new initial conditions. The continuous time method is used for solving inside a particular window. A figure of this method can be found here 32 for a hypothetical 1D problem. Learning rate decay is restarted after each time window. We note that a similar approach can be found here [32].

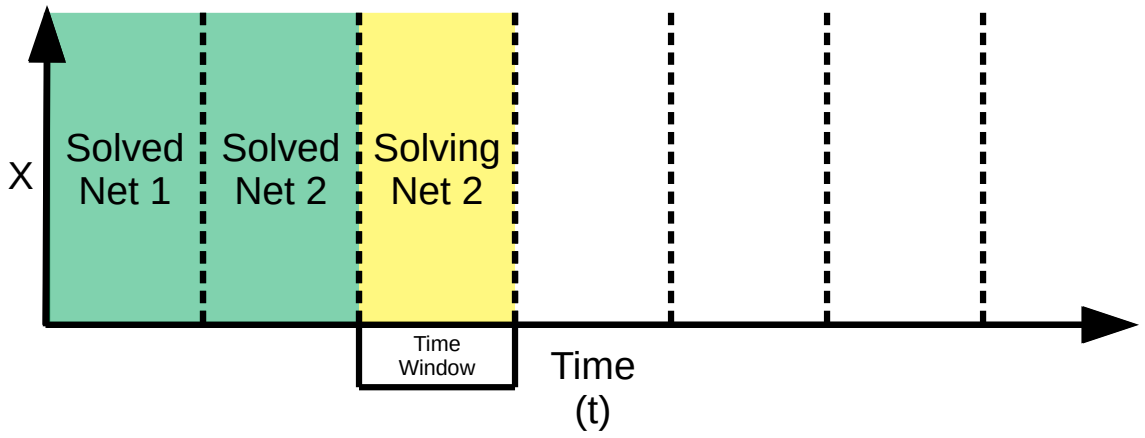


Figure 32: Moving Time Window Method.

6.3 Case Setup

The case setup for this problem is similar to many of the previous tutorials however we have 2 key differences. In this example, we show how to set up a sequence of domains to iteratively solve for. We also create custom Modulus nodes to enforce periodicity in the solution.

Note: The python script for this problem can be found at `examples/taylor_green/`.

6.3.1 Sequence of Train Domains

First we will construct our geometry similar to the previous problems.

```
# Parameters for the domain
channel_length = (0.0, 2*np.pi)
channel_width = (0.0, 2*np.pi)
channel_height = (0.0, 2*np.pi)

# define geometry
rec = Box((channel_length[0], channel_width[0], channel_height[0]),
          (channel_length[1], channel_width[1], channel_height[1]))
geo = rec

# define sympy variables to parametrize domain curves
x, y, z = Symbol('x'), Symbol('y'), Symbol('z')

# kinematic viscosity
nu = 0.002

# param range
# time window size
time_window_size = 1.0

# time domain
t_symbol = Symbol('t')
time_range = (0, time_window_size)
param_ranges = {t_symbol: time_range}
```

Listing 25: Defining Geometry

We are also defining values for how large our time window will be. In this case we will solve to 1 unit of time. Now we will define a train domain for solving the first time window. The initial conditions are coming from the above equations.

```
class ICTrain(TrainDomain):
```

```
  name = 'initial_conditions'
  nr_iterations = 1

  def __init__(self, **config):
    super(ICTrain, self).__init__()
    # ic
    ic = geo.interior_bc(outvar_sympy={'u': sin(x)*cos(y)*cos(z),
                                       'v': -cos(x)*sin(y)*cos(z),
                                       'w': 0,
                                       'p': 1.0/16 * (cos(2*x)+cos(2*y))*(cos(2*z)+2)},
                         batch_size_per_area=8,
                         bounds={x: channel_length,
                                 y: channel_width,
                                 z: channel_height},
                         lambda_sympy={'lambda_u': 10,
                                       'lambda_v': 10,
                                       'lambda_w': 10,
                                       'lambda_p': 10},
                         param_ranges={t_symbol: 0})
    self.add(ic, name="ic")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0,
                                             'momentum_x': 0,
                                             'momentum_y': 0,
                                             'momentum_z': 0},
                               bounds={x: channel_length,
                                       y: channel_width,
                                       z: channel_height},
                               batch_size_per_area=8,
                               param_ranges=param_ranges)
    self.add(interior, name="Interior")
```

Listing 26: Defining Train domain for first time window

Notice that we now name the time domain and give a value for number of iterations. This will tell the solver what file to save results in as well as how many cycles to train on this domain. Now we define the domain for all subsequent time windows,

```
class IterativeTrain(TrainDomain):
  name = 'iteration'
  nr_iterations = 19

  def __init__(self, **config):
    super(IterativeTrain, self).__init__()
    # ic
    ic = geo.interior_bc(outvar_sympy={'u_ic': 0,
                                       'v_ic': 0,
                                       'w_ic': 0,
                                       'p_ic': 0},
                         batch_size_per_area=8,
                         bounds={x: channel_length,
                                 y: channel_width,
                                 z: channel_height},
                         lambda_sympy={'lambda_u_ic': 10,
                                       'lambda_v_ic': 10,
                                       'lambda_w_ic': 10,
                                       'lambda_p_ic': 10},
                         param_ranges={t_symbol: 0})
    self.add(ic, name="IterativeIC")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0,
                                             'momentum_x': 0,
                                             'momentum_y': 0,
                                             'momentum_z': 0},
                               bounds={x: channel_length,
                                       y: channel_width,
                                       z: channel_height},
                               batch_size_per_area=8,
                               param_ranges=param_ranges)
    self.add(interior, name="IterativeInterior")
```

Listing 27: Defining Train domain for subsequent time windows

In this train domain we have a new initial condition that will come from our previous time window. We will solve this domain for 19 windows putting our total simulation time at 20 including the first time window. The values `u_ic` will be explained in the next section. We finish defining the domains by making an inference domain to save the results,

```
class TaylerInference(InferenceDomain):
  def __init__(self, **config):
    super(TaylerInference, self).__init__()
    # inf data time 0
    res = 50
    mesh_x, mesh_y, mesh_z = np.meshgrid(np.linspace(0, 2*np.pi, res),
                                         np.linspace(0, 2*np.pi, res),
                                         np.linspace(0, 2*np.pi, res),
                                         indexing='ij')
    mesh_x = np.expand_dims(mesh_x.flatten(), axis=-1)
    mesh_y = np.expand_dims(mesh_y.flatten(), axis=-1)
    mesh_z = np.expand_dims(mesh_z.flatten(), axis=-1)
    for i, specific_t in enumerate(np.linspace(time_range[0], time_window_size, 5)):
      interior = {'x': mesh_x,
                  'y': mesh_y,
                  'z': mesh_z,
                  't': np.full_like(mesh_x, specific_t)}
      inf = Inference(interior, ['u','v','w','p','shifted_t'])
      self.add(inf, "Inference_"+str(i).zfill(4))

    # inf data time 0
    res = 256
    mesh_x, mesh_y = np.meshgrid(np.linspace(0, 2*np.pi, res),
                                 np.linspace(0, 2*np.pi, res),
                                 indexing='ij')
    mesh_x = np.expand_dims(mesh_x.flatten(), axis=-1)
    mesh_y = np.expand_dims(mesh_y.flatten(), axis=-1)
    mesh_z = np.zeros_like(mesh_x)
    for i, specific_t in enumerate(np.linspace(time_range[0], time_window_size, 5)):
      interior = {'x': mesh_x,
                  'y': mesh_y,
                  'z': mesh_z,
                  't': np.full_like(mesh_x, specific_t)}
      inf = Inference(interior, ['u','v','w','p','shifted_t'])
      self.add(inf, "InferencePlane_"+str(i).zfill(4))
```

Listing 28: Defining Inference Domain

6.3.2 Sequence Solver

Now we will define the solver similar to the other problems. We make several user defined Modulus nodes to handle the periodic boundaries and the time stepping. First, we make a node that shifts the `t` to the current time window being solved. After each iteration we will add a value to this tensorflow variable causing this shift. Next we make a node that subtracts the solution from the previous time window with the current and labels it `u_ic, v_ic, w_ic, p_i` . This is then used to define the initial condition constraint in the previous train domain. We then make a node that computes a sin and cos embedding of the spatial inputs. This embedding is then feed to the neural network. Doing this ensures the network will give a periodic solution every 2π. Finally, we define the networks. We now see that the input to the network is this shifted time and spatial embedding. There are two networks constructed. The first gives our solution we are solving for and the other stores the solution from the previous time window.

```
class TaylerGreenSolver(Solver):
  seq_train_domain = [ICTrain, IterativeTrain]
  iterative_train_domain = IterativeTrain
  inference_domain = TaylerInference

  def __init__(self, **config):
    super(TaylerGreenSolver, self).__init__(**config)

    # make time window that moves
    self.time_window = tf.get_variable("time_window", [],
                                       initializer=tf.constant_initializer(0),
                                       trainable=False,
                                       dtype=tf.float32)
    def slide_time_window(invar):
      outvar = Variables()
      outvar['shifted_t'] = invar['t'] + self.time_window
      return outvar

    # make node for difference between velocity and the previous time window of velocity
    def make_ic_loss(invar):
```

```
    outvar = Variables()
    outvar['u_ic'] = invar['u'] - tf.stop_gradient(invar['u_prev_step'])
    outvar['v_ic'] = invar['v'] - tf.stop_gradient(invar['v_prev_step'])
    outvar['w_ic'] = invar['w'] - tf.stop_gradient(invar['w_prev_step'])
    outvar['p_ic'] = invar['p'] - tf.stop_gradient(invar['p_prev_step'])
    return outvar

  # make node periodic boundary
  def make_periodic_boundary(invar):
    outvar = Variables()
    outvar['x_sin'] = tf.sin(invar['x'])
    outvar['x_cos'] = tf.cos(invar['x'])
    outvar['y_sin'] = tf.sin(invar['y'])
    outvar['y_cos'] = tf.cos(invar['y'])
    outvar['z_sin'] = tf.sin(invar['z'])
    outvar['z_cos'] = tf.cos(invar['z'])
    return outvar

  self.equations = (NavierStokes(nu=nu, rho=1, dim=3, time=True).make_node()
                + [Node(make_periodic_boundary)]
                + [Node(make_ic_loss)]
                + [Node(slide_time_window)])

  flow_net = self.arch.make_node(name='flow_net',
                                 inputs=['x_sin', 'x_cos',
                                         'y_sin', 'y_cos',
                                         'z_sin', 'z_cos',
                                         'shifted_t'],
                                 outputs=['u',
                                          'v',
                                          'w',
                                          'p'])
  flow_net_prev_step = self.arch.make_node(name='flow_net_prev_step',
                                           inputs=['x_sin', 'x_cos',
                                                   'y_sin', 'y_cos',
                                                   'z_sin', 'z_cos',
                                                   'shifted_t'],
                                           outputs=['u_prev_step',
                                                    'v_prev_step',
                                                    'w_prev_step',
                                                    'p_prev_step'])
  self.nets = [flow_net, flow_net_prev_step]

@classmethod
def update_defaults(cls, defaults):
  defaults.update({
      'network_dir': './network_checkpoint_taylor_green_re_200',
      'layer_size': 256,
      'max_steps': 100000,
      'decay_steps': 1000,
      'xla': True,
  })
```

Listing 29: Defining Sequence Solver

When we solve the domains sequentially we need do some work to move the time window and keep track of network weights. We can do this by creating the following method which creates a TensorFlow operation. This operation will be called after solving each window. Here we update the shifted time window, restart the global step for the learning rate decay, and store the weights for the next time window.

```
def custom_update_op(self):
  # zero train step op
  global_step = [v for v in tf.get_collection(tf.GraphKeys.VARIABLES) if 'global_step' in v.name][0]
  zero_step_op = tf.assign(global_step, tf.zeros_like(global_step))

  # make update op that shifts time window
  update_time = tf.assign_add(self.time_window, time_window_size)

  # make update op that sets weights from_flow_net to flow_net_prev_step
  prev_assign_step = []
  flow_net_variables = [v for v in tf.trainable_variables() if 'flow_net/' in v.name]
  flow_net_prev_step_variables = [v for v in tf.trainable_variables() if 'flow_net_prev_step' in v.name]
  for v, v_prev_step in zip(flow_net_variables, flow_net_prev_step_variables):
    prev_assign_step.append(tf.assign(v_prev_step, v))
  prev_assign_step = tf.group(*prev_assign_step)

  return tf.group(update_time, zero_step_op, prev_assign_step)
```

Listing 30: Defining Train Domain

Finally we will start our solver with,

```
if __name__ == '__main__':
  ctr = ModulusController(TaylerGreenSolver)
  ctr.run()
```

Listing 31: Defining Train Domain

We note here that the way we solve iterative domains is very general. We will see in subsequent chapters that we can use this structure to implement a complex iterative algorithm to solve conjugate heat transfer problems.

6.4 Results and Post-processing

After solving this problem we can visualize the results. If we look in our network directory we should see,

```
current_step.txt initial_conditions iteration_0000 iteration_0001...
```

If we look in any of these directories we see the typical files storing network checkpoint and results. Plotting at time 15 gives the snapshot 31. To validate these results we have conducted the same simulation using a spectral solver. Comparing the point-wise error of transient simulations can be misleading as this is a chaotic dynamical system and any small errors will quickly cause large differences. Instead we look at the average turbulent kinetic energy decay (TKE) of the simulation. A figure of this can be found here, 34.

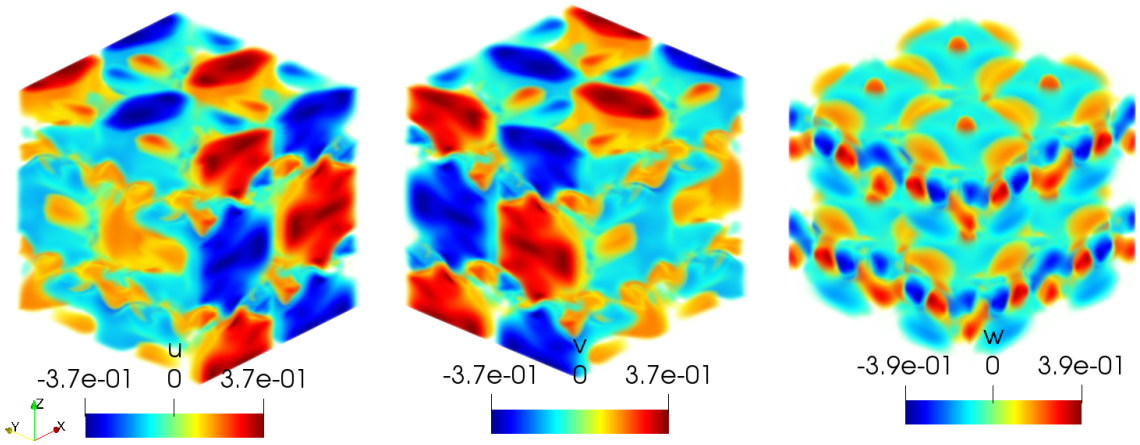


Figure 33: Taylor-Green vortex at time 15.0.

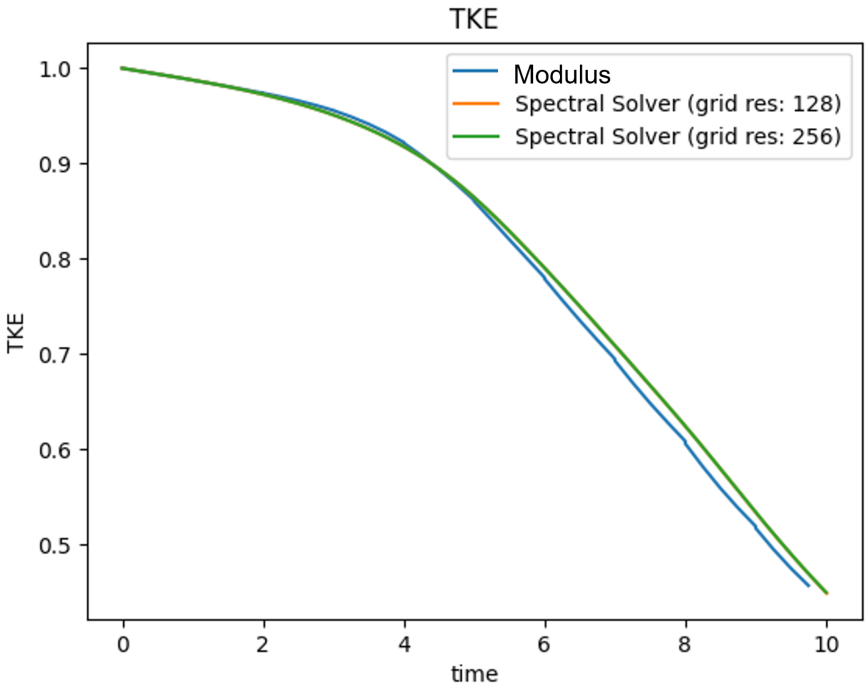


Figure 34: Taylor-Green Trubulent kinetic energy decay.

7 Ordinary Differential Equations: Coupled Spring Mass system

7.1 Introduction

In this tutorial we will use Modulus to solve a system of coupled ordinary differential equations. Since the APIs used for this problem are already covered in previous tutorial, we will only focus on the problem description without going into the details of the code.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity flow and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you to refer to tutorial 4 for information on defining new differential equations, and solving time dependent problems in Modulus.

7.2 Problem Description

In this tutorial, we will solve a simple spring mass system as shown in Figure 35. The systems shows three masses attached to each other by four springs. The springs slide along a friction-less horizontal surface. The masses are assumed to be point masses and the springs are mass-less. We will solve the problem such that the masses ($m's$) and the spring constants ($k's$) are constants, but they can later be paramterized if we intend to solve the parameterized problem (Tutorial 16).

The model's equations are given as below:

$$\begin{aligned} m_1 x_1''(t) &= -k_1 x_1(t) + k_2(x_2(t) - x_1(t)), \\ m_2 x_2''(t) &= -k_2(x_2(t) - x_1(t)) + k_3(x_3(t) - x_2(t)), \\ m_3 x_3''(t) &= -k_3(x_3(t) - x_2(t)) - k_4 x_3(t). \end{aligned} \tag{74}$$

Where, $x_1(t), x_2(t)$, and $x_3(t)$ denote the mass positions along the horizontal surface measured from their equilibrium position, plus right and minus left. As shown in the figure, first and the last spring are fixed to the walls.

For this tutorial, we will assume the following conditions:

$$\begin{aligned} [m_1, m_2, m_3] &= [1, 1, 1], \\ [k_1, k_2, k_3, k_4] &= [2, 1, 1, 2], \\ [x_1(0), x_2(0), x_3(0)] &= [1, 0, 0], \\ [x_1'(0), x_2'(0), x_3'(0)] &= [0, 0, 0]. \end{aligned} \tag{75}$$

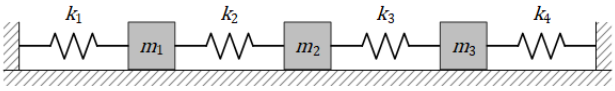


Figure 35: Three masses connected by four springs on a friction-less surface

7.3 Case Setup

The case setup for this problem is very similar to the one we followed in the tutorial 4. We will define the differential equations in `spring_mass_ode.py` and then define the domain and the solver in `spring_mass_solver.py`.

Note: The python scripts for this problem can be found at `examples/ode_spring_mass/`.

7.3.1 Defining the Equations

The equations of the system (74) can be coded using the sympy notation similar to tutorial 4.

```
from sympy import Symbol, Function, Number
from modulus.pdes import PDES
from modulus.variables import Variables

class SpringMass(PDES):
```

```
  name= 'SpringMass'

  def __init__(self, k=(2, 1, 1, 2), m=(1, 1, 1)):

    self.k  = k
    self.m  = m

    k1 = k[0]
    k2 = k[1]
    k3 = k[2]
    k4 = k[3]
    m1 = m[0]
    m2 = m[1]
    m3 = m[2]

    t=Symbol('t')
    input_variables = {'t':t}

    x1 = Function('x1')(*input_variables)
    x2 = Function('x2')(*input_variables)
    x3 = Function('x3')(*input_variables)

    if type(k1) is str:
        k1 = Function(k1)(*input_variables)
    elif type(k1) in [float, int]:
        k1 = Number(k1)
    if type(k2) is str:
        k2 = Function(k2)(*input_variables)
    elif type(k2) in [float, int]:
        k2 = Number(k2)
    if type(k3) is str:
        k3 = Function(k3)(*input_variables)
    elif type(k3) in [float, int]:
        k3 = Number(k3)
    if type(k4) is str:
        k4 = Function(k4)(*input_variables)
    elif type(k4) in [float, int]:
        k4 = Number(k4)

    if type(m1) is str:
        m1 = Function(m1)(*input_variables)
    elif type(m1) in [float, int]:
        m1 = Number(m1)
    if type(m2) is str:
        m2 = Function(m2)(*input_variables)
    elif type(m2) in [float, int]:
        m2 = Number(m2)
    if type(m3) is str:
        m3 = Function(m3)(*input_variables)
    elif type(m3) in [float, int]:
        m3 = Number(m3)

    self.equations = Variables()
    self.equations['ode_x1'] = m1*(x1.diff(t)).diff(t) + k1*x1 - k2*(x2 - x1)
    self.equations['ode_x2'] = m2*(x2.diff(t)).diff(t) + k2*(x2 - x1) - k3*(x3 - x2)
    self.equations['ode_x3'] = m3*(x3.diff(t)).diff(t) + k3*(x3 - x2) + k4*x3
```

Listing 32: Defining the Ordinary Differential Equations

Here, we wrote each parameter ($k's$and $m's$) as function and substitute it as a number if its constant. This will allow us to parameterize any of this constants by passing them as a string.

7.3.2 Solving the ODEs: Creating Geometry, defining ICs and making the Neural Network Solver

Once we have the ODEs defined, we can easily form the train domain as seen in earlier tutorials. In this example, we will use `Point1D` geometry to create the point mass. We will also define the time range of the solution and create symbol for time (t) to define the initial condition etc. in train domain. The below code shows the geometry definition for this problem.

```
from sympy import Symbol, Eq
import numpy as np

from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain
from modulus.data import Validation
from modulus.sympy_utils.geometry_1d import Point1D
from modulus.controller import ModulusController
```

```
from modulus.plot_utils.vtk import var_to_vtk

from spring_mass_ode import SpringMass

# define time variable and range
t_max = 10.0
t_symbol = Symbol('t')
time_range = {t_symbol: (0, t_max)}

geo = Point1D(0)
```

Listing 33: Generating Geometry

Next, we will define the train domain using the 1d point we just defined. Please note that we would not be using the x-coordinate (x) information of the point, and it is only used to sample a point in space. We will assign it different values for variable (t) only (initial conditions and ODEs over the time-range).

The code to define the train domain is shown below:

```
class SpringMassTrain(TrainDomain):
  def __init__(self, **config):
    super(SpringMassTrain, self).__init__()

    # initial conditions
    IC = geo.boundary_bc(outvar_sympy={'x1': 1.,
                                       'x2': 0,
                                       'x3': 0,
                                       'x1__t': 0,
                                       'x2__t': 0,
                                       'x3__t': 0},
                         param_ranges={t_symbol: 0},
                         batch_size_per_area=1)
    self.add(IC, name="IC")

    # solve over given time period
    interior = geo.boundary_bc(outvar_sympy={'ode_x1': 0.0,
                                             'ode_x2': 0.0,
                                             'ode_x3': 0.0},
                               param_ranges=time_range,
                               batch_size_per_area=500)
    self.add(interior, name="Interior")
```

Listing 34: Defining training domain

Next we will define the validation data for this problem. The solution of this problem can be obtained analytically and the expression can be coded into dictionaries of numpy arrays for x_1, x_2, and x_3. This part of the code is similar to the tutorial 4.

```
class SpringMassVal(ValidationDomain):
   def __init__(self, **config):
     super(SpringMassVal, self).__init__()
     deltaT = 0.001
     t = np.arange(0, t_max, deltaT)
     t = np.expand_dims(t, axis=-1)
     invar_numpy = {'t': t}
     outvar_numpy = {'x1': (1/6)*np.cos(t) + (1/2)*np.cos(np.sqrt(3)*t) + (1/3)*np.cos(2*t),
                     'x2': (2/6)*np.cos(t) + (0/2)*np.cos(np.sqrt(3)*t) - (1/3)*np.cos(2*t),
                     'x3': (1/6)*np.cos(t) - (1/2)*np.cos(np.sqrt(3)*t) + (1/3)*np.cos(2*t)}
     val = Validation.from_numpy(invar_numpy, outvar_numpy)
     self.add(val, name="Val")
```

Listing 35: Define Validation data

Now that we have the definitions for the training data and the validation data complete, we can form the solver and run the problem. The code to do the same can be found below:

```
class SpringMassSolver(Solver):
  train_domain = SpringMassTrain
  val_domain = SpringMassVal

  def __init__(self, **config):
    super(SpringMassSolver, self).__init__(**config)

    self.equations = SpringMass(k=(2, 1, 1, 2), m=(1, 1, 1)).make_node()
```

```
    spring_net = self.arch.make_node(name='spring_net',
                                     inputs=['t'],
                                     outputs=['x1','x2','x3'])
    self.nets = [spring_net]

  @classmethod # Explain This
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_spring_mass',
        'max_steps': 10000,
        'decay_steps': 100,
        'nr_layers': 6,
        'layer_size': 256,
        'xla': True,
        })

if __name__ == '__main__':
  ctr = ModulusController(SpringMassSolver)
  ctr.run()
```

Listing 36: Defining the equation parameters and making the Neural Network solver

Once the python file is setup, we can solve the problem by executing the solver script `spring_mass_solver.py` as seen in other tutorials.

7.4 Results and Post-processing

The results for the Modulus simulation are compared against the analytical validation data. We can see that the solution converges to the analytical result in less than a minute. The plots can be created using the .npz files that are created in the `val_domain/` directory in the network checkpoint.

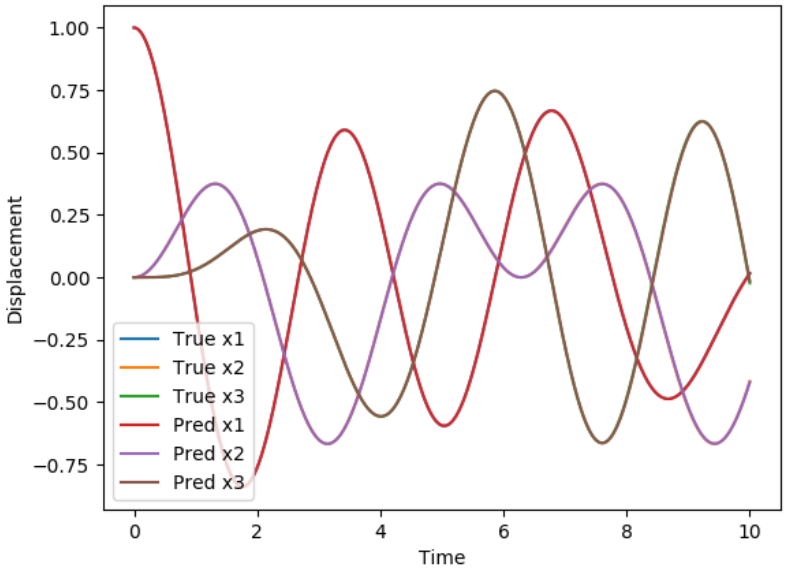


Figure 36: Comparison of Modulus results with analytical solution

8 Scalar Transport: 2D Advection Diffusion

8.1 Introduction

In this tutorial, we will use an advection-diffusion transport equation for temperature along with the Continuity and Navier-Stokes equation to model the heat transfer in a 2D flow. In this tutorial you would learn the following:

1. How to implement advection-diffusion for a scalar quantity.
2. How to create custom profiles for boundary conditions and to set up gradient boundary conditions.
3. How to use additional constraints like `IntegralContinuity` to speed up convergence.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity flow and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you to refer to tutorial 3 for information on creating monitor and inference domains.

8.2 Problem Description

In this tutorial, we will solve the heat transfer from a 3-fin heat sink. The problem describes a hypothetical scenario wherein a 2D slice of the heat sink is simulated as shown in the figure. The heat sinks are maintained at a constant temperature of 350 K and the inlet is at 293.498 K. The channel walls are treated as adiabatic. The inlet is assumed to be a parabolic velocity profile with 1.5 m/s as the peak velocity. The kinematic viscosity ν is set to 0.01 m^2/s and the Prandtl number is 5. Although the flow is laminar, the Zero Equation turbulence model is kept on.

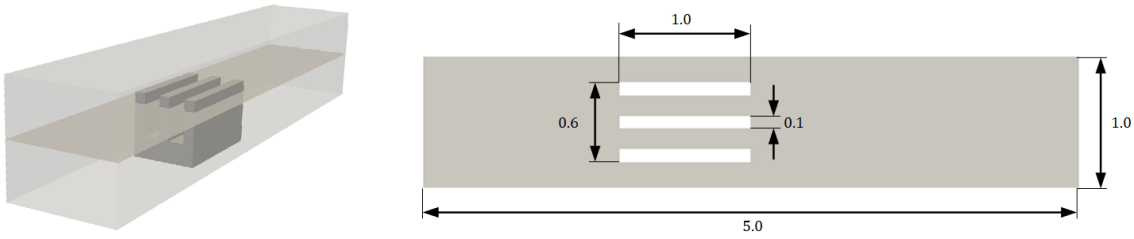


Figure 37: 2d slice of three fin heat sink geometry (All dimensions in m)

8.3 Case Setup

Note: The python script for this problem can be found at `examples/three_fin_2d/heat_sink.py`.

Importing the required packages

In this tutorial we will make use of `Channel2D` geometry to make the duct. `Line` geometry would be used to make inlet, outlet and intermediate planes for integral boundary conditions. The `AdvectionDiffusion` equation is imported from the `PDES` package. The `parabola` and `GradNormal` are imported from appropriate packages/modules to generate the required boundary conditions.

```
from sympy import Symbol
import tensorflow as tf

from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain, InferenceDomain, MonitorDomain
from modulus.data import Monitor, Inference, Validation
from modulus.sympy_utils.functions import parabola
from modulus.sympy_utils.geometry_2d import Rectangle, Channel2D, Line
from modulus.csv_utils.csv_rw import csv_to_dict
from modulus.PDES.navier_stokes import NavierStokes, GradNormal, IntegralContinuity
from modulus.PDES.turbulence_zero_eq import ZeroEquation
```

```
from modulus.PDES.advection_diffusion import AdvectionDiffusion
from modulus.node import Node
from modulus.controller import ModulusController
```

Listing 37: Importing required packages

8.3.1 Creating Geometry

For generating the geometry of this problem, we will be using `Channel2D` for duct and `Rectangle` for generating the heat sink. The way of defining `Channel2D` is same as `Rectangle`. The difference between channel and rectangle is, a channel is infinite and composed of only two curves and a rectangle is composed of four curves that form a closed boundary.

`Line` is defined using the x and y coordinates of the two endpoints and the normal direction of the curve. Note that the `Line` requires the x coordinates of both the points to be same. A line in arbitrary orientation can then be created by rotating the `Line` object.

The following code generates the geometry for the 2d heat sink problem.

```
# params for domain
channel_length = (-2.5, 2.5)
channel_width = (-0.5, 0.5)
heat_sink_origin = (-1, -0.3)
nr_heat_sink_fins = 3
gap = 0.15 + 0.1
heat_sink_length = 1.0
heat_sink_fin_thickness = 0.1
inlet_vel = 1.5
heat_sink_temp = 350
base_temp = 293.498
effective_nu = 0.01

# define geometry
channel = Channel2D((channel_length[0], channel_width[0]),
                    (channel_length[1], channel_width[1]))
heat_sink = Rectangle(heat_sink_origin,
                      (heat_sink_origin[0]+heat_sink_length, heat_sink_origin[1]+heat_sink_fin_thickness))
for i in range(1, nr_heat_sink_fins):
  heat_sink_origin = (heat_sink_origin[0], heat_sink_origin[1]+gap)
  fin = Rectangle(heat_sink_origin,
                  (heat_sink_origin[0]+heat_sink_length, heat_sink_origin[1]+heat_sink_fin_thickness))
  heat_sink = heat_sink + fin
geo = channel - heat_sink
inlet = Line((channel_length[0], channel_width[0]),
             (channel_length[0], channel_width[1]), -1)
outlet = Line((channel_length[1], channel_width[0]),
              (channel_length[1], channel_width[1]), 1)
plane1 = Line((channel_length[0]+0.5, channel_width[0]),
              (channel_length[0]+0.5, channel_width[1]), 1)
plane2 = Line((channel_length[0]+1.0, channel_width[0]),
              (channel_length[0]+1.0, channel_width[1]), 1)
plane3 = Line((channel_length[0]+3.0, channel_width[0]),
              (channel_length[0]+3.0, channel_width[1]), 1)
plane4 = Line((channel_length[0]+3.5, channel_width[0]),
              (channel_length[0]+3.5, channel_width[1]), 1)
```

Listing 38: Generating Geometry

8.3.2 Defining the Boundary conditions and Equations to solve

The boundary conditions described in the problem statement are implemented in the code shown below. Keys `'normal_gradient_c'` is used to set the gradient boundary conditions. Note, a new variable c is defined for the solving the advection diffusion equation.

$$c = T_{actual} - T_{inlet} \tag{76}$$

In addition to the continuity and Navier Stokes equations in 2D, advection diffusion equation (equation 77) with no source term is solved in the `Interior`. The thermal diffusivity D for this problem is 0.002 m^2/s.

$$uc_x + vc_y = D(c_{xx} + c_{yy}) \tag{77}$$

Integral continuity planes are created by specifying the targeted mass flow rate through these planes. For a parabolic velocity of 1.5 m/s, the integral mass flow is 1 which is added as an additional constraint to speed up the convergence. Similar to tutorial 2 we will define keys for `'integral_continuity'` on the plane boundaries and set its value to `1` to specify the targeted mass flow. These planes (lines for 2d geometry) would then be used when the `IntegralContinuity` equation class is called in the solver section of the code.

The parabolic profile can be created by using the `parabola` function by specifying the variable for sweep, the two intercepts and max height.

```
class HeatSinkTrain(TrainDomain):
  def __init__(self, **config):
    super(HeatSinkTrain, self).__init__()
    x, y = Symbol('x'), Symbol('y')

    # inlet
    parabola_sympy = parabola(y, inter_1=channel_width[0], inter_2=channel_width[1], height=inlet_vel)
    inletBC = inlet.boundary_bc(outvar_sympy={'u': parabola_sympy, 'v': 0, 'c': 0},
                                batch_size_per_area=64)
    self.add(inletBC, name="Inlet")

    # inlet
    outletBC = outlet.boundary_bc(outvar_sympy={'p': 0},
                                  batch_size_per_area=64)
    self.add(outletBC, name="Outlet")

    # no slip heat_sink
    heatSinkWall = heat_sink.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'c': heat_sink_temp-base_temp},
                                         batch_size_per_area=64)
    self.add(heatSinkWall, name="HeatSinkWall")

    # no slip channel
    channelWall = channel.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'normal_gradient_c': 0},
                                      batch_size_per_area=256)
    self.add(channelWall, name="ChannelWall")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     advection_diffusion': 0},
                               bounds={x: (channel_length), y: (channel_width)},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                                             'lambda_momentum_x': geo.sdf,
                                             'lambda_momentum_y': geo.sdf,
                                             'lambda_advection_diffusion': 1.0},
                               batch_size_per_area=1000)
    self.add(interior, name="Interior")

    # integral continuity
    plane1Cont = plane1.boundary_bc(outvar_sympy={'integral_continuity': 1},
                                    batch_size_per_area=64,
                                    lambda_sympy={'lambda_integral_continuity': 0.1})
    plane2Cont = plane2.boundary_bc(outvar_sympy={'integral_continuity': 1},
                                    batch_size_per_area=64,
                                    lambda_sympy={'lambda_integral_continuity': 0.1})
    plane3Cont = plane3.boundary_bc(outvar_sympy={'integral_continuity': 1},
                                    batch_size_per_area=64,
                                    lambda_sympy={'lambda_integral_continuity': 0.1})
    plane4Cont = plane4.boundary_bc(outvar_sympy={'integral_continuity': 1},
                                    batch_size_per_area=64,
                                    lambda_sympy={'lambda_integral_continuity': 0.1})
    outletCont = outlet.boundary_bc(outvar_sympy={'integral_continuity': 1},
                                    batch_size_per_area=64,
                                    lambda_sympy={'lambda_integral_continuity': 0.1})
    self.add(plane1Cont, name="IntegralContinuity1")
    self.add(plane2Cont, name="IntegralContinuity2")
    self.add(plane3Cont, name="IntegralContinuity3")
    self.add(plane4Cont, name="IntegralContinuity4")
    self.add(outletCont, name="IntegralContinuity5")
```

Listing 39: Defining boundary conditions and equations to solve

8.3.3 Creating Monitors, Inference and Validation domains

This process is very similar to earlier tutorials and hence we won't cover this in detail. We will make use of `InferenceDomain`, `MonitorDomain` and `ValidationDomain` for this problem. The validation data comes from a 2D simulation computed using OpenFOAM.

```
# validation data
mapping = {'Points:0': 'x', 'Points:1': 'y',
           'U:0': 'u', 'U:1': 'v', 'p': 'p', 'nuT': 'nu', 'T': 'c'}
openfoam_var = csv_to_dict('openfoam/heat_sink_zeroEq_Pr5_mesh20.csv', mapping)
openfoam_var['nu'] += effective_nu
openfoam_var['c'] += -base_temp
openfoam_invar_numpy = {key: value for key, value in openfoam_var.items() if key in ['x', 'y']}
openfoam_outvar_numpy = {key: value for key, value in openfoam_var.items() if key in ['u', 'v', 'p', 'nu', 'c']}


class HeatSinkInference(InferenceDomain):
  def __init__(self, **config):
    super(HeatSinkInference, self).__init__()
    x, y = Symbol('x'), Symbol('y')

    # save entire domain
    interior = Inference(geo.sample_interior(10000, bounds={x: (channel_length), y: (channel_width)}),
                         ['u', 'v', 'p', 'nu', 'c', 'normal_distance', 'normal_distance__y', '
     normal_distance__x', 'nu__x', 'nu__y'])
    self.add(interior, name="Inference")

class HeatSinkVal(ValidationDomain):
  def __init__(self, **config):
    super(HeatSinkVal, self).__init__()
    val = Validation.from_numpy(openfoam_invar_numpy, openfoam_outvar_numpy)
    self.add(val, name='Val')

class HeatSinkMonitor(MonitorDomain):
  def __init__(self, **config):
    super(HeatSinkMonitor, self).__init__()
    x, y = Symbol('x'), Symbol('y')

    # metric for mass imbalance, momentum imbalance and peak velocity magnitude
    global_monitor = Monitor(geo.sample_interior(100, bounds={x: (channel_length), y: (channel_width)}),
                             {'mass_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['continuity']))
     ,
                              'momentum_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_x'])+tf.abs(var['momentum_x'])),
                              'pressure_drop': lambda var: tf.reduce_max(var['p'])
                              })
    self.add(global_monitor, 'GlobalMonitor')

    # metric on the fins
    force = Monitor(heat_sink.sample_boundary(100),
                    {'force_x': lambda var: tf.reduce_sum(var['normal_x']*var['area']*var['p']),
                     'force_y': lambda var: tf.reduce_sum(var['normal_y']*var['area']*var['p'])})
    self.add(force, 'Force')

    # metric at outlet
    peak_T = Monitor(outlet.sample_boundary(100),
                     {'peakT': lambda var: tf.reduce_max(var['c'])})
    self.add(peak_T, 'Temp')
```

Listing 40: Define Validation, Inference and Monitor domains

8.3.4 Making the Neural Network Solver

For this problem, we will make two separate network architectures for solving flow and heat parts. This is just to demonstrate how the networks can be separated to increase accuracy.

Additional equations (compared to tutorial 3) for `IntegralContinuity`, `AdvectionDiffusion` and `GradNormal` are called and the variable to compute is defined for the `GradNormal` and `AdvectionDiffusion` boundary are defined.

Also, it is possible to stop gradient calls on a particular equation if one wishes to decouple it from rest of the equations. This uses the `tf.stop_gradient` in the backend to stop the computation of gradients https://www.tensorflow.org/api_docs/python/tf/stop_gradient. In this problem, we stop the gradient calls on u, v. This prevents the network from optimizing u, and v to minimize the residual from the advection equation. In this way, we can make the system one-way coupled, where the heat does not influence the flow but the flow influences the heat.

```
class HeatSinkSolver(Solver):
  train_domain = HeatSinkTrain
  val_domain = HeatSinkVal
  inference_domain = HeatSinkInference
  monitor_domain = HeatSinkMonitor
```

```
  def __init__(self, **config):
    super(HeatSinkSolver, self).__init__(**config)

    self.equations = (NavierStokes(nu='nu', rho=1.0, dim=2, time=False).make_node()
                      + ZeroEquation(nu=effective_nu, dim=2, max_distance=0.5).make_node()
                      + AdvectionDiffusion(T='c', rho=1.0, D=0.01/5, dim=2, time=False).make_node(
     stop_gradients=['u', 'v'])
                      + [Node.from_sympy(geo.sdf, 'normal_distance')]
                      + IntegralContinuity().make_node()
                      + GradNormal('c', dim=2, time=False).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    heat_net = self.arch.make_node(name='heat_net',
                                   inputs=['x', 'y'],
                                   outputs=['c'])
    self.nets = [flow_net, heat_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_heat_sink',
        'max_steps': 500000,
        'decay_steps': 5000,
        })

if __name__ == '__main__':
  ctr = ModulusController(HeatSinkSolver)
  ctr.run()
```

Listing 41: Defining the equation parameters and making the Neural Network solver

8.4 Running the Modulus solver

Once the python file is setup, you can save the file and exit out of the text editor. In the command prompt, type in:

```
python heat_sink.py
```

8.5 Results and Post-processing

The results for the Modulus simulation are compared against the OpenFOAM data in figure 38.

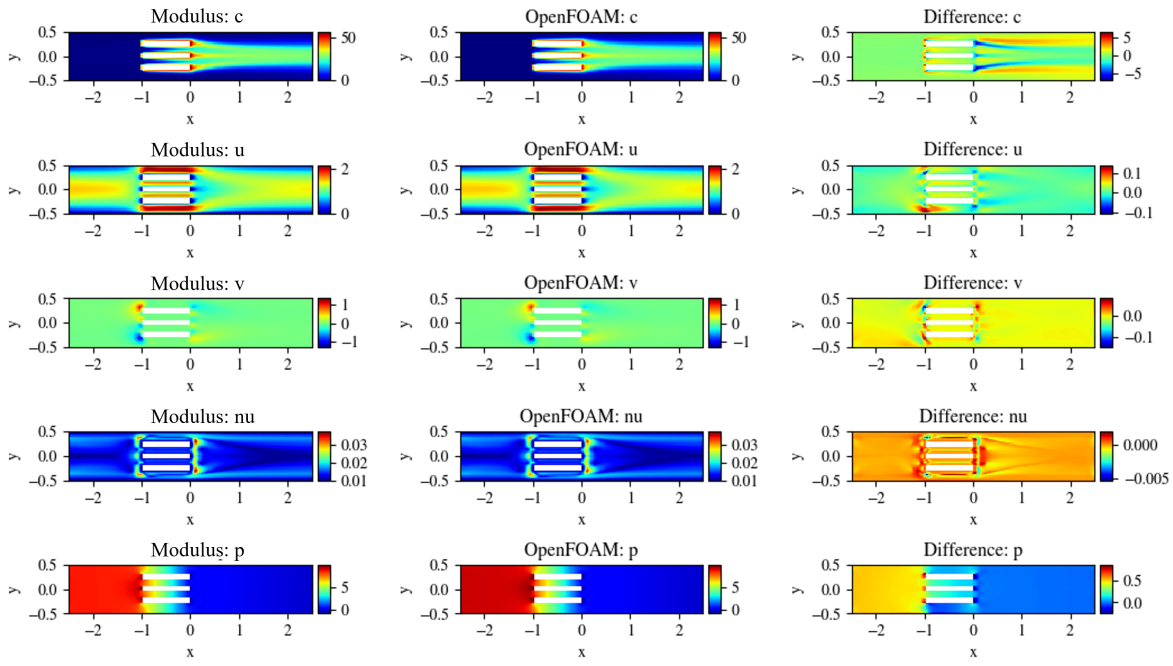


Figure 38: Left: Modulus. Center: OpenFOAM. Right: Difference

9 Interface problem by Variational method

9.1 Introduction

In this tutorial, we will walk through the process of solving a PDE using the variational formulation. We will show how to use variational method to solve the interface PDE problem using Modulus. The use of variational method (weak formulation) also allows us to handle problems with point source with ease and we will cover in this tutorial too. Thus, in this tutorial you would learn the following:

1. How to solve a PDE in its variational form (continuous and discontinuous) in Modulus.
2. How to generate test functions and their derivative data on desired point sets.
3. How to use quadrature in the Modulus.
4. How to solve a problem with a point source (Dirac Delta function).

Prerequisites

This tutorial assumes that assumes that you have completed tutorial 2 on Lid Driven Cavity and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you to refer Section 1.7 from the Theory chapter for more details on weak solutions of PDEs.

Note: All the scripts referred in this tutorial can be found in `examples/discontinuous_galerkin/`. The examples in this chapter also use the `quadpy` library which is not pre-installed in the Modulus docker container. Please install it manually using the below command.

```
pip install quadpy
```

9.2 Problem Description

In this tutorial, we will solve the Poisson equation with Dirichlet boundary conditions. The problem represents an interface between two domains. Let $\Omega_1 = (0, 0.5) \times (0, 1)$, $\Omega_2 = (0.5, 1) \times (0, 1)$, $\Omega = (0, 1)^2$. The interface is $\Gamma = \overline{\Omega}_1 \cap \overline{\Omega}_2$, and the Dirichlet boundary is $\Gamma_D = \partial\Omega$. The domain for the problem can be visualized in the Figure 39. The problem was originally defined in [25].

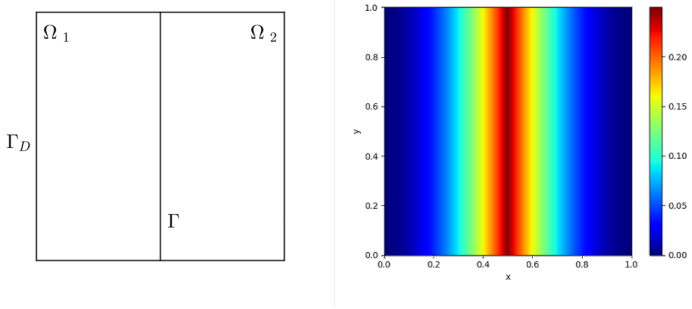


Figure 39: Left: Domain of interface problem. Right: True Solution

The PDEs for the problem are defined as

$$\begin{cases} -\Delta u = f & \text{in } \Omega_1 \cup \Omega_2, \quad (78) \\ u = g_D & \text{on } \Gamma_D, \quad (79) \\ \left[\dfrac{\partial u}{\partial \mathbf{n}}\right] = g_I & \text{on } \Gamma, \quad (80) \end{cases}$$

where $f = -2$, $g_I = 2$ and

$$g_D = \begin{cases} x^2 & 0 \le x \le \frac{1}{2} \\ (x-1)^2 & \frac{1}{2} < x \le 1 \end{cases}.$$

The g_D is the exact solution of (78)-(80).

The jump $[\cdot]$ on the interface Γ is defined by

$$\left[\frac{\partial u}{\partial \mathbf{n}}\right] = \nabla u_1 \cdot \mathbf{n}_1 + \nabla u_2 \cdot \mathbf{n}_2, \tag{81}$$

where u_i is the solution in Ω_i and the $\mathbf{n}_i$ is the unit normal on $\partial\Omega_i \cap \Gamma$.

As suggested in the original reference, this problem does not accept a strong (classical) solution but only a unique weak solution (g_D) which is shown in Figure 39.

Note: Please be advised that, in the original paper [25], the PDE is incorrect and (78)-(80) defines the corrected PDEs for the problem.

9.3 Variational Form

Since (80) suggests that the solution's derivative is broken at interface (Γ) , we have to do the variational form on Ω_1 and Ω_2 separately. Equations 82 and 83 show the continuous and discontinuous variational formulation for the above problem. For brevity, we will only give the final variational forms here. For the detailed derivation of these formulations, we recommend you to refer Theory Appendix B.

Variational form for Continuous type formulation :

$$\int_\Omega (\nabla u \cdot \nabla v - fv)dx - \int_\Gamma g_I v ds - \int_{\Gamma_D} \frac{\partial u}{\partial \mathbf{n}} v ds = 0 \tag{82}$$

Variational form for Discontinuous type formulation :

$$\sum_{i=1}^{2} (\nabla u_i \cdot v_i - f v_i)dx - \sum_{i=1}^{2} \int_{\Gamma_D} \frac{\partial u_i}{\partial \mathbf{n}} v_i ds - \int_\Gamma (g_I \langle v \rangle + \langle \nabla u \rangle [\![v]\!])ds = 0 \tag{83}$$

In the following subsections, we will see how to implement these variational forms in the Modulus.

9.4 Continuous type formulation

In this subsection, we introduce how to implement the continuous type variational form (82) in Modulus. The code for this example can be found in `dg_pinns.py`.

Like all the other examples, we first import all the packages needed

```
from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain, InferenceDomain
from modulus.data import Validation, Inference
from modulus.sympy_utils.geometry_2d import Rectangle, Line
from modulus.PDES.diffusion import Diffusion
from modulus.controller import ModulusController
from modulus.variables import Variables
from modulus.vpinn_utils.test_functions import Test_Function, Legendre_test, Trig_test
from modulus.vpinn_utils.integral import tensor_int
```

Listing 42: Importing required packages

9.4.1 Creating the Geometry

Since we have an interface in the middle of the domain, we will define the geometry by left and right parts separately. This will allow us to capture the interface information by sampling on the boundary that is common to the two halves.

```
# define geometry
rec_1 = Rectangle((0, 0), (0.5, 1))
rec_2 = Rectangle((0.5, 0), (1, 1))
rec = rec_1 + rec_2
```

Listing 43: Defining domain

In this example, we recommend to use the variational form in conjunction with traditional PINNs. The PINNs' loss is essentially a point-wise residual, and the loss function performs good for a smooth solution. Hence, we impose the traditional PINNs' loss for areas away from boundaries and interfaces.

9.4.2 Defining the Boundary conditions and Equations to solve

With the geometry defined for the problem we define the `TrainDomain` as shown in the below code.

```
class DGTrain(TrainDomain):
  def __init__(self, **config):
    super(DGTrain, self).__init__()

    # walls
    Wall = rec_1.boundary_bc(outvar_sympy={'u': x**2},
                             batch_size_per_area=2000,
                             criteria=x<0.5)
    self.add(Wall, name="Wall1")
    Wall = rec_2.boundary_bc(outvar_sympy={'u': (x-1)**2},
                             batch_size_per_area=2000,
                             criteria=x>0.5)
    self.add(Wall, name="Wall2")

    # interior
    interior = rec.interior_bc(outvar_sympy={'diffusion_u': -2},
                               bounds={x: (0, 1), y: (0, 1)},
                               lambda_sympy={'lambda_diffusion_u': Abs(x-0.5)},
                               batch_size_per_area=10000,
                               fixed_var=False,
                               quasirandom=True)
    self.add(interior, name="Interior")

    # center line
    center = rec_1.boundary_bc(outvar_sympy={},
                               criteria=Eq(0.5, x),
                               batch_size_per_area=2000,
                               fixed_var=False,
                               quasirandom=True)
    self.add(center, name="CenterLine")
```

Listing 44: Defining TrainDomain

There are four parts in this class: `Wall1`, `Wall2`, `Interior`, an `CenterLine`, as indicated by their names in the train domain. The `Wall1` and `Wall2` are used to impose the Dirichlet boundary condition (79). Even though the problem is mainly solved by variational form, the Dirichlet boundary condition has to be imposed as we have seen in the regular PINNs. This is because the Dirichlet boundary condition does not appear in the variational form (82). Also, this is the exact reason why it named as enforced boundary condition in Finite Element Method (FEM).

The `Interior` is used to impose the traditional PINNs' loss, so it is similar to the other examples. The -2 in the `outvar_sympy` is the right-hand side term in (78).

The `Interior` and the `CenterLine` are used to define the variational loss as well. In these two parts, we only give the geometric information and leave the `outvar_sympy` as blank. By doing this, Modulus will generate relative training points on those geometries so that we can use them later to form loss functions that uses the points of combined domains. Since we will use these training points to calculate the integral value, the quasi-random sampling is the best choice as it has higher accuracy than uniform random points. To enable this, we set `quasirandom=True` . Moreover, we want to use all points to train and do not want these points changed over time because that will change the error of integral, so we set `fixed_var=False`.

9.4.3 Creating the Validation Domains

The `ValidationDomain` is created by the code shown below. The process of defining the validation data from analytical solution is similar to the example covered in tutorial 4.

```
class DGVal(ValidationDomain):
  def __init__(self, **config):
    super(DGVal, self).__init__()
    # make validation data
    delta_x = 0.01
    delta_y = 0.01
    x0 = np.arange(0, 1, delta_x)
    y0 = np.arange(0, 1, delta_y)
    x_grid, y_grid = np.meshgrid(x0, y0)
    x_grid = np.expand_dims(x_grid.flatten(), axis=-1)
    y_grid = np.expand_dims(y_grid.flatten(), axis=-1)
    u = np.where(x_grid<=0.5, x_grid**2, (x_grid-1)**2)
    invar_numpy = {'x': x_grid, 'y': y_grid}
    outvar_numpy = {'u': u}
```

```
    val = Validation.from_numpy(invar_numpy, outvar_numpy)
    self.add(val, name='Val')
```

Listing 45: Defining ValidationDomain and InferenceDomain

9.4.4 Creating the Variational Loss and Solver

Since we have both, traditional PINN and variational loss in this problem, we will have to implement them both in our code. For the traditional PINNs' loss, we need to add an equation node in the Solver. In this example, we are using `Diffusion`. This procedure is similar to the previous examples. In this code, we call the solution as `'u'`.

For the variational loss, we use the class function `custom_loss` . The basic idea is to first get the points' coordinates, and then get the prediction and necessary derivatives on those points. Then, choose the set of test functions and then do a similar procedure. We can then form our variational loss.

Let's look at (82) term by term. What we need is ∇u and ∇v in `'Interior`, f and v in `Interior`, v on `CenterLine`, and ∇u on `Wall1` and `Wall2`, where v is the test function . In the code, we may use the domain's name as the key to get the geometric information. For example, to get the coordinates of interior points and its average area, we may use:

```
x_interior = domain_invar['Interior']['x']
y_interior = domain_invar['Interior']['y']
area_interior = domain_invar['Interior']['area']
```

Listing 46: Getting the coordinates of interior points and its average area

To get the prediction of u and ∇u on those points, we can use the following code:

```
u_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['u']
ux_interior = tf.gradients(u_interior, x_interior)[0]
uy_interior = tf.gradients(u_interior, y_interior)[0]
```

Listing 47: Getting the predictions and their derivatives on desired points

Besides, the unit normal can be accessed by using keys `normal_x` and `normal_y`. For example, the following code will fetch the training points and geometric information of the outside wall:

```
# get points on outside wall
x_outside = tf.concat([domain_invar['Wall1']['x'], domain_invar['Wall2']['x']], axis=0)
y_outside = tf.concat([domain_invar['Wall1']['y'], domain_invar['Wall2']['y']], axis=0)
normal_x_outside = tf.concat([domain_invar['Wall1']['normal_x'], domain_invar['Wall2']['normal_x']], axis=0)
normal_y_outside = tf.concat([domain_invar['Wall1']['normal_y'], domain_invar['Wall2']['normal_y']], axis=0)
area_outside = tf.concat([domain_invar['Wall1']['area'], domain_invar['Wall2']['area']], axis=0)
```

Listing 48: Getting the training points and geometric information of outside walls

To generate the test functions, we can make use of the `Test_Function` class . In Modulus, Legendre, 1st and 2nd kind of Chebyshev polynomials and trigonometric functions are already implemented as the test functions and can be selected directly. You can also define your own test functions by providing its name, domain, and SymPy expression in `meta_test_function` class. In the `Test_Function`, you will need to provide a dictionary of the name and order of the test functions (`name_ord_dict` in the parameter list), the upper and lower bound of your domain (`box` in the parameter list), and what kind of derivatives you will need (`diff_list` in the parameter list). For example, if v_{xxy} is needed, we may add `[1,1,2]` in the `diff_list`. There are shortcuts for `diff_list`. If we need all the components of gradient of test function, we may add `'grad'` in `diff_list`, and if the Laplacian of the test function is needed, we may add `'Delta'`. The `box` parameter if left unspecified, is set to the default values, i.e. for Legendre polynomials $[-1, 1]^n$, for trignometric functions $[0, 1]^n$, etc.

Complex value functions are also accepted by setting `is_real = False` in `meta_test_function` class. In this case, the combination of real and imaginary parts of the functions will be used as the test functions. In our problem, we are using Legendre polynomials up to degree 9, and trigonometric functions $e^{i\pi(k_1 x + k_2 y)}$ with a frequency no more than 5. This can be done by

```
v = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
Trig_test: [k for k in range(5)]}, diff_list=['grad'])
```

Listing 49: Making test functions

To evaluate the test functions, we may use `eval_test` method as below

```
v_interior = v.eval_test('v', x_interior, y_interior)
vx_interior = v.eval_test('vx', x_interior, y_interior)
vy_interior = v.eval_test('vy', x_interior, y_interior)
v_outside = v.eval_test('v', x_outside, y_outside)
v_center = v.eval_test('v', x_center, y_center)
```

Listing 50: Evaluating test functions

Now, all the resulting variables, like `v_interior`, are N by M tensors, where N is the number of points, and M is the number of the test functions.

To form the integration, we can use the `tensor_int` function in the Modulus. This function has three parameters `w`, `v`, and `u`. The `w` is the quadrature weight for the integration. For uniform random points or quasi-random points, it is precisely the average area. The `v` is an N by M tensor, and `u` is a 1 by M tensor. If `u` is provided, this function will return a 1 by M tensor, and each entry is $\int_\Omega uv_i dx$, for $i = 1, \cdots, M$. If `u` is not provided, it will return a 1 by M tensor, and each entry is $\int_\Omega v_i dx$, for $i = 1, \cdots, M$. To code to form the integral is the following

```
f = -2. + tf.zeros_like(x_interior)        # RHS: f=-2.
uxvx = ux_interior*vx_interior
uyvy = uy_interior*vy_interior
fv = f*v_interior
int_interior = tensor_int(area_interior, uxvx+uyvy-fv)
int_center = tensor_int(area_center, 2.0*v_center)      # 2.0 is the jump on the interface.
int_outside = tensor_int(area_outside, v_outside, dudn)
```

Listing 51: Forming integrals

The whole code to make the Solver is the following:

```
class DGSolver(Solver):
  train_domain = DGTrain
  val_domain = DGVal

  def __init__(self, **config):
    super(DGSolver, self).__init__(**config)

    self.equations = (Diffusion(T='u', D=1, dim=2, time=False).make_node())

    diffusion_net = self.arch.make_node(name='diffusion_net',
                                        inputs=['x', 'y'],
                                        outputs=['u'])
    self.nets = [diffusion_net]

  def custom_loss(self, domain_invar, pred_domain_outvar, true_domain_outvar, step):
    # get points on interior of rec
    x_interior = domain_invar['Interior']['x']
    y_interior = domain_invar['Interior']['y']
    area_interior = domain_invar['Interior']['area']

    # compute u for interior of rec
    u_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['u']
    ux_interior = tf.gradients(u_interior, x_interior)[0]
    uy_interior = tf.gradients(u_interior, y_interior)[0]

    # get points on center line in rec (this normal is going in the positive direction)
    x_center = domain_invar['CenterLine']['x']
    y_center = domain_invar['CenterLine']['y']
    area_center = domain_invar['CenterLine']['area']

    # get points on outside wall
    x_outside = tf.concat([domain_invar['Wall1']['x'], domain_invar['Wall2']['x']], axis=0)
    y_outside = tf.concat([domain_invar['Wall1']['y'], domain_invar['Wall2']['y']], axis=0)
    normal_x_outside = tf.concat([domain_invar['Wall1']['normal_x'], domain_invar['Wall2']['normal_x']], axis=0)

    normal_y_outside = tf.concat([domain_invar['Wall1']['normal_y'], domain_invar['Wall2']['normal_y']], axis=0)

    area_outside = tf.concat([domain_invar['Wall1']['area'], domain_invar['Wall2']['area']], axis=0)

    # compute u, ux, uy for outside wall
    u_outside = self.nets[0].evaluate({'x': x_outside, 'y': y_outside})['u']
    ux_outside = tf.gradients(u_outside, x_outside)[0]
    uy_outside = tf.gradients(u_outside, y_outside)[0]
    dudn = normal_x_outside*ux_outside + normal_y_outside*uy_outside
```

```
    # discontinuous galerkin
    v = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
                                     Trig_test: [k for k in range(5)]}, diff_list=['grad'])
    v_keep = Test_Function(name_ord_dict={Legendre_test: [k for k in range(2)],
                                     Trig_test: [k for k in range(2)]}, diff_list=['grad'])
    name_dict = {'v': [[x_interior, y_interior], [x_outside, y_outside], [x_center, y_center]], 'vx': [[
     x_interior, y_interior]], 'vy': [[x_interior, y_interior]]}
    v_val = v.fetch_batch(name_dict=name_dict, batch_size=50, keep_test=v_keep)
    v_interior = v_val['v'][0]
    vx_interior = v_val['vx'][0]
    vy_interior = v_val['vy'][0]
    v_outside = v_val['v'][1]
    v_center = v_val['v'][2]
    f = -2. + tf.zeros_like(x_interior)        # RHS: f=-2. Refactor in the future
    uxvx = ux_interior*vx_interior
    uyvy = uy_interior*vy_interior
    fv = f*v_interior
    int_interior = tensor_int(area_interior, uxvx+uyvy-fv)
    int_center = tensor_int(area_center, 2.0*v_center)      # 2.0 is the jump on the interface. Refactor in the
      future
    int_outside = tensor_int(area_outside, v_outside, dudn)

    # make loss function and return it
    loss = Variables()
    loss['loss_variational'] = tf.reduce_sum(tf.square(int_interior-int_center-int_outside)) # add variational
     loss
    return loss

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_dg_pinns',
        'max_steps': 100000,
        'decay_steps': 1000,
        'layer_size': 512,
        'xla': True,
        })
```

Listing 52: Code for Solver

9.4.5 Results and Post-processing

The results for the problem are shown in Figure 40.

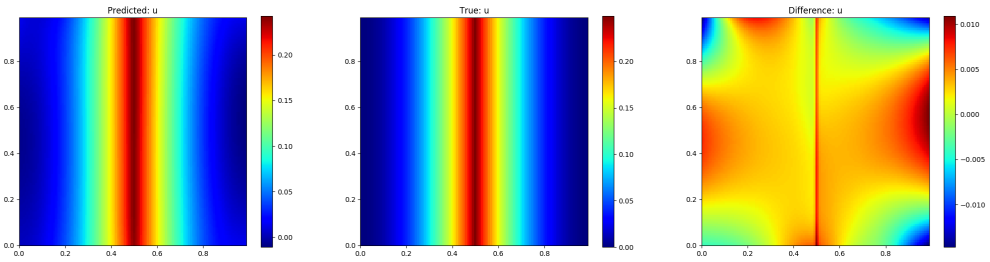


Figure 40: Left: Modulus. Center: Analytical. Right: Difference.

9.5 Discontinuous type formulation

In this subsection, we will solve the discontinuous type formulation (83), where u is approximated by two neural networks and v is defined on different domains. Although the continuous method imposes the solution's continuity automatically (because it is approximated by one neural network, which is continuous), the discontinuous method gives the solver more freedom because the solution is not necessarily continuous and smooth on the interface. For the sake of simplicity, we only emphasize the main change of the code versus the last example. The code can be found in dg_pinns_2nn.py.

9.5.1 Defining the Boundary conditions and Equations

Since we are using two neural networks, we will define the PINNs' loss for each neural network on different domains. Hence, the `TrainDomain` will be defined in the following way:

```
class DGTrain(TrainDomain):
  def __init__(self, **config):
    super(DGTrain, self).__init__()

    # walls
    Wall = rec_1.boundary_bc(outvar_sympy={'u_1': x**2},
                             batch_size_per_area=2000,
                             criteria=~Eq(x, 0.5),
                             fixed_var=False,
                             quasirandom=True)
    self.add(Wall, name="Wall1")
    Wall = rec_2.boundary_bc(outvar_sympy={'u_2': (x-1)**2},
                             batch_size_per_area=2000,
                             criteria=~Eq(x, 0.5),
                             fixed_var=False,
                             quasirandom=True)
    self.add(Wall, name="Wall2")

    # interior
    interior = rec_1.interior_bc(outvar_sympy={'diffusion_u_1': -2},
                                 bounds={x: (0, 1), y: (0, 1)},
                                 lambda_sympy={'lambda_diffusion_u_1': 1},
                                 batch_size_per_area=10000,
                                 fixed_var=False,
                                 quasirandom=True)
    self.add(interior, name="Interior1")
    interior = rec_2.interior_bc(outvar_sympy={'diffusion_u_2': -2},
                                 bounds={x: (0, 1), y: (0, 1)},
                                 lambda_sympy={'lambda_diffusion_u_2': 1},
                                 batch_size_per_area=10000,
                                 fixed_var=False,
                                 quasirandom=True)
    self.add(interior, name="Interior2")

    # line in center
    center = rec_1.boundary_bc(outvar_sympy={'diffusion_interface_dirichlet_u_1_u_2': 0,
                                             'diffusion_interface_neumann_u_1_u_2': 0},
                               criteria=Eq(0.5, x),
                               batch_size_per_area=2000,
                               fixed_var=False,
                               quasirandom=True)
    self.add(center, name="CenterLine")
```

Listing 53: `TrainDomain`

9.5.2 Creating the Variational Loss and Solver

In the solver, we first impose the PINNs' constraints for the two neural networks. Besides, since the solution is continuous on the interface, we have to impose the interface's continuity. This will be done by using `DiffusionInterface`. The code for this part is

```
self.equations = (Diffusion(T='u_1', D=1, dim=2, time=False).make_node()
                + Diffusion(T='u_2', D=1, dim=2, time=False).make_node()
                + DiffusionInterface('u_1', 'u_2', 1.0, -1.0, dim=2, time=False).make_node()
                + [Node(lambda var: Variables({'u': tf.where(var['x'] < 0.5, var['u_1'], var['u_2'])}))])

diffusion_net_1 = self.arch.make_node(name='diffusion_net_1',
                                      inputs=['x', 'y'],
                                      outputs=['u_1'])
diffusion_net_2 = self.arch.make_node(name='diffusion_net_2',
                                      inputs=['x', 'y'],
                                      outputs=['u_2'])
self.nets = [diffusion_net_1, diffusion_net_2]
```

Listing 54: Making the PINNs' PDEs constraints and continuity

For the variational loss, we will follow the formulation in (83). Essentially, the difference is the part $\int_\Gamma \langle \nabla u \rangle [\![v]\!] ds$. To write the code, we may assume $v_1 \neq 0$ and $v_2 = 0$ and then $v_1 = 0$ and $v_2 \neq 0$. This will form two integral values and will also make the multi GPU training possible. The detailed code for the variational loss is shown below. Although the code is relatively lengthy, it is fairly self-explanatory.

```
def custom_loss(self, domain_invar, pred_domain_outvar, true_domain_outvar, step):
  # get points on interior of rec
  x_interior_1 = domain_invar['Interior1']['x']
  y_interior_1 = domain_invar['Interior1']['y']
  area_interior_1 = domain_invar['Interior1']['area']
  x_interior_2 = domain_invar['Interior2']['x']
  y_interior_2 = domain_invar['Interior2']['y']
  area_interior_2 = domain_invar['Interior2']['area']

  # compute u for interior of rec
  u_interior_1 = self.nets[0].evaluate({'x': x_interior_1, 'y': y_interior_1})['u_1']
  ux_interior_1 = tf.gradients(u_interior_1, x_interior_1)[0]
  uy_interior_1 = tf.gradients(u_interior_1, y_interior_1)[0]
  u_interior_2 = self.nets[1].evaluate({'x': x_interior_2, 'y': y_interior_2})['u_2']
  ux_interior_2 = tf.gradients(u_interior_2, x_interior_2)[0]
  uy_interior_2 = tf.gradients(u_interior_2, y_interior_2)[0]

  # get points on center line in rec (this normal is going in the positive direction)
  x_center = domain_invar['CenterLine']['x']
  y_center = domain_invar['CenterLine']['y']
  normal_x_center = domain_invar['CenterLine']['normal_x']
  normal_y_center = domain_invar['CenterLine']['normal_y']
  area_center = domain_invar['CenterLine']['area']

  # compute u for center line in rec
  u_center_1 = self.nets[0].evaluate({'x': x_center, 'y': y_center})['u_1']
  ux_center_1 = tf.gradients(u_center_1, x_center)[0]
  uy_center_1 = tf.gradients(u_center_1, y_center)[0]
  u_center_2 = self.nets[1].evaluate({'x': x_center, 'y': y_center})['u_2']
  ux_center_2 = tf.gradients(u_center_2, x_center)[0]
  uy_center_2 = tf.gradients(u_center_2, y_center)[0]
  ux_center_ave = 0.5*(ux_center_1+ux_center_2)
  uy_center_ave = 0.5*(uy_center_1+uy_center_2)
  dudn_center_ave = normal_x_center*ux_center_ave+normal_y_center*uy_center_ave

  # get points on outside wall
  x_outside_1 = domain_invar['Wall1']['x']
  y_outside_1 = domain_invar['Wall1']['y']
  normal_x_outside_1 = domain_invar['Wall1']['normal_x']
  normal_y_outside_1 = domain_invar['Wall1']['normal_y']
  area_outside_1 = domain_invar['Wall1']['area']
  x_outside_2 = domain_invar['Wall2']['x']
  y_outside_2 = domain_invar['Wall2']['y']
  normal_x_outside_2 = domain_invar['Wall2']['normal_x']
  normal_y_outside_2 = domain_invar['Wall2']['normal_y']
  area_outside_2 = domain_invar['Wall2']['area']

  # compute u, ux, uy for outside wall
  u_outside_1 = self.nets[0].evaluate({'x': x_outside_1, 'y': y_outside_1})['u_1']
  ux_outside_1 = tf.gradients(u_outside_1, x_outside_1)[0]
  uy_outside_1 = tf.gradients(u_outside_1, y_outside_1)[0]
  dudn_1 = normal_x_outside_1*ux_outside_1 + normal_y_outside_1*uy_outside_1
  u_outside_2 = self.nets[1].evaluate({'x': x_outside_2, 'y': y_outside_2})['u_2']
  ux_outside_2 = tf.gradients(u_outside_2, x_outside_2)[0]
  uy_outside_2 = tf.gradients(u_outside_2, y_outside_2)[0]
  dudn_2 = normal_x_outside_2*ux_outside_2 + normal_y_outside_2*uy_outside_2

  # discontinuous galerkin
  v_1 = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
                                     Trig_test: [k for k in range(5)]},
                      box=[[0, 0], [0.5, 1]],
                      diff_list=['grad'])
  v_2 = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
                                     Trig_test: [k for k in range(5)]},
                      box=[[0.5, 0], [1, 1]],
                      diff_list=['grad'])
  v_interior_1 = v_1.eval_test('v', x_interior_1, y_interior_1)
  vx_interior_1 = v_1.eval_test('vx', x_interior_1, y_interior_1)
  vy_interior_1 = v_1.eval_test('vy', x_interior_1, y_interior_1)
  v_outside_1 = v_1.eval_test('v', x_outside_1, y_outside_1)
  v_center_1 = v_1.eval_test('v', x_center, y_center)
  v_interior_2 = v_2.eval_test('v', x_interior_2, y_interior_2)
  vx_interior_2 = v_2.eval_test('vx', x_interior_2, y_interior_2)
  vy_interior_2 = v_2.eval_test('vy', x_interior_2, y_interior_2)
  v_outside_2 = v_2.eval_test('v', x_outside_2, y_outside_2)
  v_center_2 = v_2.eval_test('v', x_center, y_center)
  f_1 = -2. + tf.zeros_like(x_interior_1)        # RHS: f=-2. Refactor in the future
  uxvx_1 = ux_interior_1*vx_interior_1
  uyvy_1 = uy_interior_1*vy_interior_1
  fv_1 = f_1*v_interior_1
```

```
    f_2 = -2. + tf.zeros_like(x_interior_2)        # RHS: f=-2. Refactor in the future
    uxvx_2 = ux_interior_2*vx_interior_2
    uyvy_2 = uy_interior_2*vy_interior_2
    fv_2 = f_2*v_interior_2
    int_interior_1 = tensor_int(area_interior_1, uxvx_1+uyvy_1-fv_1)
    int_interior_2 = tensor_int(area_interior_2, uxvx_2+uyvy_2-fv_2)
    int_center_jump_1 = 0.5*tensor_int(area_center, 2.0*v_center_1)      # 2.0 is the jump on the interface.
     Refactor in the future
    int_center_jump_2 = 0.5*tensor_int(area_center, 2.0*v_center_2)      # 2.0 is the jump on the interface.
     Refactor in the future
    int_center_ave_1 = tensor_int(area_center, v_center_1, dudn_center_ave)
    int_center_ave_2 = tensor_int(area_center, v_center_2, -dudn_center_ave)
    int_outside_1 = tensor_int(area_outside_1, v_outside_1, dudn_1)
    int_outside_2 = tensor_int(area_outside_2, v_outside_2, dudn_2)

    # make loss function and return it
    loss_1 = tf.reduce_sum(tf.square(int_interior_1-int_center_jump_1-int_center_ave_1-int_outside_1))
    loss_2 = tf.reduce_sum(tf.square(int_interior_2-int_center_jump_2-int_center_ave_2-int_outside_2))
    loss = Variables()
    loss['loss_variational'] = loss_1 + loss_2
    return loss
```

Listing 55: Making the variational loss for discontinuous formulation

9.5.3 Results and Post-processing

The results for the problem solved using discontinuous formulation are shown in Figure 41.

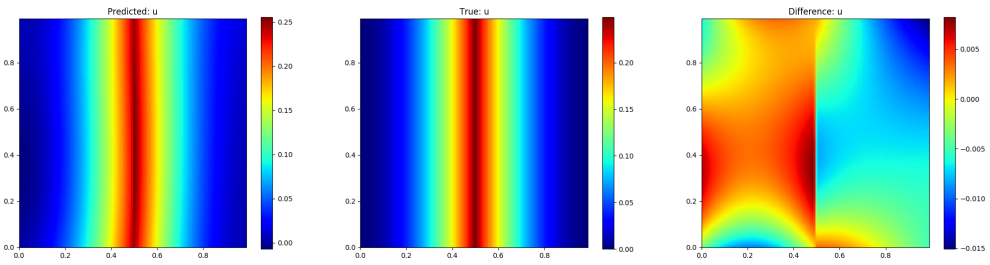


Figure 41: Left: Modulus. Center: Analytical. Right: Difference.

9.6 Quadrature

To calculate the integral more efficiently, we can use the quadrature rules. In Modulus, we have implemented some commonly used quadrature rules based on `quadpy` package (https://github.com/nschloe/quadpy). In this example, we will show how to use the quadrature feature in the Modulus. For the sake of simplicity, we will use continuous type variational formulation (82). The code can be found in `dg_pinns_quadrature.py`. For brevity, we describe only the key modifications in this subsection.

In (82), we will calculate the first and second integrals by using quadrature. So, we need quadrature in Ω and Γ. To this end, we import `Quad_Rect, Quad_Line, Quad_Collection` from `modulus.vpinn_utils.integral`.

Since we know that there is discontinuity on Γ, we split the quadrature on the left and right half of Ω. For each half, we use 20 order product Gauss quadrature. The code to do the following can be found below.

```
paras = [[[[0, 0.5], [0, 1]], 20, True, lambda n: quadpy.c2.product(quadpy.c1.gauss_legendre(n))],
    [[[0.5, 1], [0, 1]], 20, True, lambda n: quadpy.c2.product(quadpy.c1.gauss_legendre(n))]]
quad_rec = Quad_Collection(Quad_Rect, paras)
```

Listing 56: Quadrature rule for Ω

The `paras` is a list of parameters for `Quad_Rect` since we have two rectangles to calculate. Then, `Quad_Collection` will calculate the two quadrature and then union them together.

For the quadrature of the center line Γ, we use 20 order Gauss quadrature rule. And the code is just simply:

```
quad_line = Quad_Line([0.5, 0], [0.5, 1], 20)
```

Listing 57: Quadrature rule for Γ

After defining these two quadrature rules, we may have the quadrature points and weights for Ω and Γ. Thus, we no longer need these two geometries in `TrainDomain`, and it will be changed to

```
class DGTrain(TrainDomain):
  def __init__(self, **config):
    super(DGTrain, self).__init__()

    # walls
    Wall = rec_1.boundary_bc(outvar_sympy={'u': x**2},
                             batch_size_per_area=2000,
                             criteria=x<0.5,
                             fixed_var=False,
                             quasirandom=True)
    self.add(Wall, name="Wall1")
    Wall = rec_2.boundary_bc(outvar_sympy={'u': (x-1)**2},
                             batch_size_per_area=2000,
                             criteria=x>0.5,
                             fixed_var=False,
                             quasirandom=True)
    self.add(Wall, name="Wall2")

    # interior
    interior = rec.interior_bc(outvar_sympy={'diffusion_u': -2},
                               bounds={x: (0, 1), y: (0, 1)},
                               lambda_sympy={'lambda_diffusion_u': Abs(x-0.5)},
                               batch_size_per_area=10000,
                               fixed_var=False,
                               quasirandom=True)
    self.add(interior, name="Interior")
```

Listing 58: `TrainDomain` with quadrature rule

To form the integrals in the custom loss, we can fetch the quadrature points and weights by the following code:

```
x_interior = quad_rec.points_tensor[:, 0:1]
y_interior = quad_rec.points_tensor[:, 1:2]
weights_interior = quad_rec.weights_tensor

x_center = quad_line.points_tensor[:, 0:1]
y_center = quad_line.points_tensor[:, 1:2]
weights_center = quad_line.weights_tensor
```

Listing 59: Fetching the quadrature points and weights

And then, the integrals can be formed by

```
int_interior = tensor_int(weights_interior, uxvx+uyvy-fv)
int_center = tensor_int(weights_center, 2.0*v_center)
```

Listing 60: Forming the integrals

The results for the problem solved using quadrature rules for integration are shown in Figure 42.

At this point, we have solved the interface problem using 3 different methods. Table 2 compares the relative L_2 errors of each of these methods. As we can see, the difference between the continuous and discontinuous method is very close for this example. However, the discontinuous method is advantageous in certain applications as it offers more flexibility in handling the discontinuity. Also, using the quadrature rules to compute the integrals help to improve the accuracy while minimizing the computational cost.

9.7 Point source and Dirac Delta function

Weak formulation enables solution of PDEs with distributions, e.g., Dirac Delta function . The Dirac Delta function $\delta(x)$ is defined as

$$\int_{\mathbb{R}} f(x)\delta(x)dx = f(0),$$

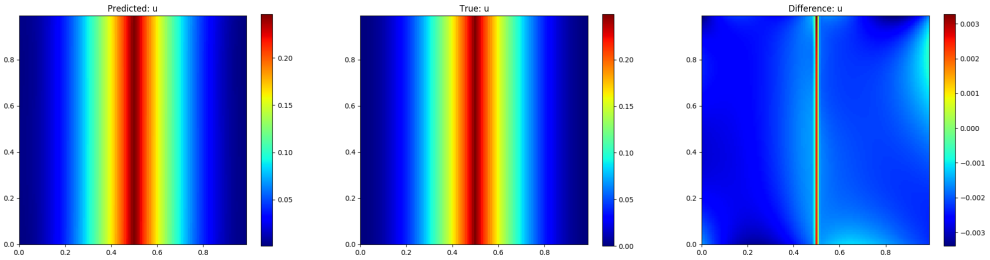


Figure 42: Left: Modulus. Center: Analytical. Right: Difference.

Table 2: Relative Error comparions of different methods on the Interface problem

Method	Relative L_2 error
Continuous	0.0329
Discontinuous	0.0480
Continuous method with Quadrature	0.0199

for all continuous compactly supported functions f.

In this subsection, we solve the following problem:

$$\begin{cases} -\Delta u = \delta & \text{in } \Omega, \quad (84) \\ u = 0 & \text{on } \partial\Omega, \quad (85) \end{cases}$$

where $\Omega = (-0.5, 0.5)^2$ (Figure 43). In physics, this means there is a point source in the middle of the domain with 0 Lebesgue measure in $\mathbb{R}^2$. The corresponding weak formulation is

$$\int_\Omega \nabla u \cdot \nabla v dx - \int_\Gamma \frac{\partial u}{\partial \mathbf{n}} v ds = v(0,0) \quad (86)$$

The code of this example can be found in dg_pinns_point_source.py.

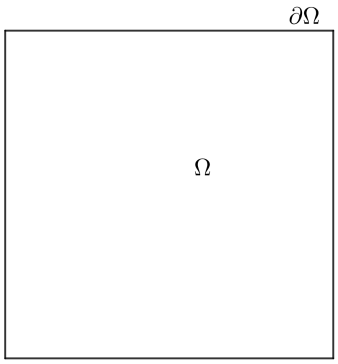


Figure 43: Domain for the point source problem.

9.7.1 Creating the Geometry

As before, we will use both the weak and differential form to solve (84)-(85) and (86). Since the solution has a sharp gradient around the origin, which will cause issues for traditional PINNs, we will weight this area lower using the lambda weighting functions. The geometry can be defined by:

```
rec = Rectangle((-0.5, -0.5), (0.5, 0.5))
```

Listing 61: Making the geometry

9.7.2 Creating the Variational Loss and Solver

As shown in (86), the only difference to the previous examples is the right-hand side term is the value of v instead of an integral. We hence only need to change the `fv` in the code as follows:

```
v_source = v.eval_test('v', tf.zeros_like(x_interior), tf.zeros_like(y_interior))
fv = v_source
```

Listing 62: Making the right hand side term

Besides, in this example, we will use the L-BFGS solver instead of Adam optimizer. L-BFGS is a second-order quasi-Newton methods, and it usually has better performance than Adam optimizer. To use the L-BFGS optimizer, we first import it by `modulus.optimizer import LBFGSOptimizer`. Then, we set `optimizer = LBFGSOptimizer` in `DGSolver`, and then set `'max_steps': 0` in the `update_defaults` function.

The whole code of the `Solver` is the following:

```
class DGSolver(Solver):
  train_domain = DGTrain
  inference_domain = DGInference
  optimizer = LBFGSOptimizer

  def __init__(self, **config):
    super(DGSolver, self).__init__(**config)

    self.equations = (Diffusion(T='u', D=1, dim=2, time=False).make_node())

    diffusion_net = self.arch.make_node(name='diffusion_net',
    inputs=['x', 'y'],
    outputs=['u'])
    self.nets = [diffusion_net]

  def custom_loss(self, domain_invar, pred_domain_outvar, true_domain_outvar):
    # get points on interior of rec
    x_interior = domain_invar['Interior']['x']
    y_interior = domain_invar['Interior']['y']
    area_interior = domain_invar['Interior']['area']

    # compute u for interior of rec
    u_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['u']
    ux_interior = tf.gradients(u_interior, x_interior)[0]
    uy_interior = tf.gradients(u_interior, y_interior)[0]


      # get points on outside wall
        x_outside = tf.concat([domain_invar['Wall1']['x'], domain_invar['Wall2']['x']], axis=0)
        y_outside = tf.concat([domain_invar['Wall1']['y'], domain_invar['Wall2']['y']], axis=0)
        normal_x_outside = tf.concat([domain_invar['Wall1']['normal_x'], domain_invar['Wall2']['normal_x']],
     axis=0)
        normal_y_outside = tf.concat([domain_invar['Wall1']['normal_y'], domain_invar['Wall2']['normal_y']],
     axis=0)
        area_outside = tf.concat([domain_invar['Wall1']['area'], domain_invar['Wall2']['area']], axis=0)

    # get points on outside wall
    x_outside = domain_invar['OutsideWall']['x']
    y_outside = domain_invar['OutsideWall']['y']
    normal_x_outside = domain_invar['OutsideWall']['normal_x']
    normal_y_outside = domain_invar['OutsideWall']['normal_y']
    area_outside = domain_invar['OutsideWall']['area']

    # compute u, ux, uy for outside wall
    u_outside = self.nets[0].evaluate({'x': x_outside, 'y': y_outside})['u']
    ux_outside = tf.gradients(u_outside, x_outside)[0]
    uy_outside = tf.gradients(u_outside, y_outside)[0]
    dudn = normal_x_outside*ux_outside + normal_y_outside*uy_outside


    # put code here for doing discontinuous galerkin here
    v = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
    Trig_test: [k for k in range(5)]},
    box=[[-0.5, -0.5], [0.5, 0.5]],
    diff_list=['grad'])

    vx_interior = v.eval_test('vx', x_interior, y_interior)
    vy_interior = v.eval_test('vy', x_interior, y_interior)
    v_outside = v.eval_test('v', x_outside, y_outside)
```

```
    v_source = v.eval_test('v', tf.zeros_like(x_interior), tf.zeros_like(y_interior))

    fv = v_source
    uxvx = ux_interior*vx_interior
    uyvy = uy_interior*vy_interior
    int_interior = tensor_int(area_interior, uxvx+uyvy)-fv
    int_outside = tensor_int(area_outside, v_outside, dudn)

    # make loss function and return it
    loss = Variables()
    loss['loss_variational'] = tf.reduce_sum(tf.square(int_interior-int_outside)) #tf.zeros(shape=[]) # put
     loss here
    return loss

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
      'network_dir': './network_checkpoint_dg_pinns_point_source',
      'max_steps': 0,
      'decay_steps': 1000,
      'layer_size': 512,
      'xla': True,
      })
```

Listing 63: Making the Solver

9.7.3 Results and Post-processing

The results for the problem are shown in Figure 44.

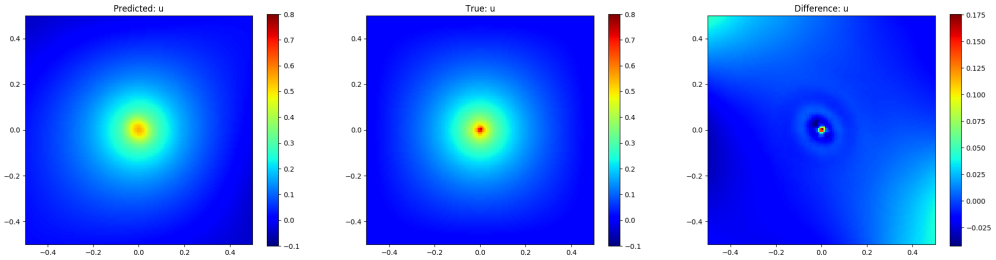


Figure 44: Left: Modulus. Center: Analytical. Right: Difference.

10 Electromagnetics: Frequency Domain Maxwell's Equation

10.1 Introduction

In this tutorial we will introduce how to use Modulus to do the electromagnetic (EM) simulation . Currently, Modulus offers the following features for frequency domain EM simulation:

1. Frequency domain Maxwell's equation in scalar form. This is same to Helmholtz equation. (This is available for 1D, 2D, and 3D. Only real form is available now.)
2. Frequency domain Maxwell's equation in vector form. (This is available for 3D case and only real form is available now.)
3. Perfect electronic conductor (PEC) boundary conditions for 2D and 3D cases.
4. Radiation boundary condition (or, absorbing boundary condition) for 3D.
5. 1D waveguide port solver for 2D waveguide source.

We will solve two electromagnetics problems in the tutorial. All the simulations are appropriately non-dimensionalized.

Note: All the scripts referred in this tutorial can be found in `examples/waveguide/`.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity and have familiarized yourself with the basics of the Modulus user interface.

10.2 Problem 1: 2D Waveguide Cavity

Let us consider a 2D domain $\Omega = [0, 2] \times [0, 2]$ as shown in Figure 45. The whole domain is vacuum. Say, relative permittivity $\epsilon_r = 1$. The left boundary is a waveguide port while the right boundary is absorbing boundary (or ABC). The top and the bottom is PEC.

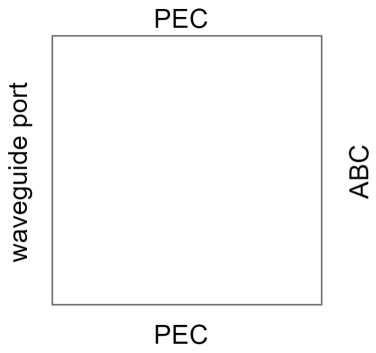


Figure 45: Domain of 2D waveguide

In this example we will solve this waveguide problem by transverse-magnetic (TM_z) mode, so that our unknown variable is $E_z(x, y)$. The governing equation in Ω is

$$\Delta E_z(x, y) + k^2 E_z(x, y) = 0,$$

where k is the wavenumber. Notice in 2D scalar case, the PEC and ABC will be simplified in the following form, respectively:

$$E_z(x, y) = 0 \text{ on top and bottom boundaries, } \frac{\partial E_z}{\partial y} = 0 \text{ on right boundary.}$$

10.2.1 Case Setup

In this subsection, we will show how to use Modulus to setup the EM solver. Similar to our previous tutorials, we first import the necessary libraries.

```
from sympy import Symbol, pi, sin, Number, Eq
from sympy.logic.boolalg import Or
from modulus.solver import Solver
from modulus.dataset import TrainDomain, InferenceDomain, ValidationDomain
from modulus.data import Inference, Validation
from modulus.csv_utils.csv_rw import csv_to_dict
from modulus.sympy_utils.geometry_2d import Rectangle
from modulus.PDES.wave_equation import HelmholtzEquation
from modulus.PDES.navier_stokes import GradNormal
from modulus.controller import ModulusController
from modulus.architecture.modified_fourier_net import ModifiedFourierNetArch
```

Listing 64: Import libraries for EM simulation

Then, we define the variables for `sympy` symbolic calculation and domain geometry.

```
x, y = Symbol('x'), Symbol('y')
# params for domain
height = 2
width = 2
# define geometry
rec = Rectangle((0, 0), (width, height))
```

Listing 65: Define `sympy` variables and geometry.

Before we define the main classes for Modulus, we will need to compute the eigenmode for waveguide solver. Since the material is uniform (vacuum), we know the closed form of the eigenmode is of form $\sin(\frac{k\pi y}{L})$, where L is the length of the port, and $k = 1, 2, \cdots$. We then define the waveguide port directly by using `sympy` function:

```
eigenmode = [1]
wave_number = 32.    # wave_number = freq/c
waveguide_port = Number(0)
for k in eigenmode:
  waveguide_port += sin(k*pi*y/height)
```

Listing 66: Define the waveguide port

To validate the result, we import the `csv` files for the validation domain below.

```
# validation data
mapping = {'x': 'x', 'y': 'y', 'u': 'u'}
validation_var = csv_to_dict('validation/2Dwaveguide_16_1.csv', mapping)
validation_invar_numpy = {key: value for key, value in validation_var.items() if key in ['x', 'y']}
validation_outvar_numpy = {key: value for key, value in validation_var.items() if key in ['u']}
```

Listing 67: Import walidation data

Now, we can define the `TrainDomain`. The BCs are defined based on the explanations provided above. Please note that in the interior domain, the weights of the PDE is `1.0/wave_number**2`. This is because when wavenumber is large, the PDE loss will be very large in the beginning and will potentially break the training. We have found that by this weighting method we can eliminate this phenomenon.

```
class HemholtzTrain(TrainDomain):
  def __init__(self, **config):
    super(HemholtzTrain, self).__init__()

    #walls
    Wall = rec.boundary_bc(outvar_sympy={'u': 0.},
                           criteria=Or(Eq(y, 0), Eq(y, height)),
                           lambda_sympy={'lambda_u': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="PEC")

    Wall = rec.boundary_bc(outvar_sympy={'u': waveguide_port},
                           criteria=Eq(x, 0),
                           lambda_sympy={'lambda_u': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="Waveguide_port")

    Wall = rec.boundary_bc(outvar_sympy={'normal_gradient_u': 0.},
```

```
                            criteria=Eq(x, width),
                            lambda_sympy={'lambda_normal_gradient_u': 10.},
                            fixed_var=False,
                            batch_size_per_area=100)
    self.add(Wall, name="ABC")

    # interior
    interior = rec.interior_bc(outvar_sympy={'helmholtz': 0.},
                               bounds={x: (0, width), y: (0, height)},
                               lambda_sympy={'lambda_helmholtz': 1.0/wave_number**2},
                               fixed_var=False,
                               batch_size_per_area=1000)
    self.add(interior, name="Interior")
```

Listing 68: Training domain for the 2D Waveguide Cavity

Validation domain has been implemented to verify the results.

```
class WaveguideVal(ValidationDomain):
  def __init__(self, **config):
    super(WaveguideVal, self).__init__()
    val = Validation.from_numpy(validation_invar_numpy, validation_outvar_numpy)
    self.add(val, name='Val')
```

Listing 69: Validation domain for the 2D Waveguide Cavity

Inference domain has been implemented to plot the results.

```
class WaveguideInference(InferenceDomain):
  def __init__(self, **config):
    super(WaveguideInference, self).__init__()
    interior_points = rec.sample_interior(100000, bounds={x: (0, width), y: (0, height)})
    inf = Inference(interior_points, ['u'])
    self.add(inf, name='Inf'+str(int(wave_number)))
```

Listing 70: Inference domain for the 2D Waveguide Cavity

For wave simulation, since the result is always periodic, Fourier feature will greatly helpful for the convergence and accuracy. The frequency of the Fourier feature can be implied by the wavenumber. The below block of code shows the solver setup.

```
class WaveguideSolver(Solver):
    train_domain = HemholtzTrain
    inference_domain = WaveguideInference
    val_domain = WaveguideVal
    arch = ModifiedFourierNetArch

  def __init__(self, **config):
    super(WaveguideSolver, self).__init__(**config)

    self.equations = (HelmholtzEquation(u='u', k=wave_number, dim=2).make_node()
                      + GradNormal(T='u', dim=2, time=False).make_node())

    self.arch.frequencies = ('axis,diagonal', [i / 2. for i in range(int(wave_number)*2+1)])
    self.arch.frequencies_params = ('axis,diagonal', [i / 2. for i in range(int(wave_number)*2+1)])

    wave_net = self.arch.make_node(name='wave_net',
                                   inputs=['x', 'y'],
                                   outputs=['u'])
    self.nets = [wave_net]

    @classmethod
    def update_defaults(cls, defaults):
      defaults.update({
          'network_dir': './network_checkpoint_waveguide2D_TMz',
          'start_lr': 1e-3,
          'decay_steps': 10000,
          'max_steps': 500000,
          'layer_size': 512,
          'nr_layers': 6,
          'xla': True,
          })

if __name__ == '__main__':
  ctr = ModulusController(WaveguideSolver)
  ctr.run()
```

Listing 71: Solver for the 2D Waveguide Cavity

10.3 Problem 2: 2D Dielectric slab waveguide

In this section, we do the 2D waveguide simulation with a dielectric slab. The problem setup is almost same as before except there is a horizontal dielectric slab in the middle of the domain. The domain is shown in Figure 46. In the

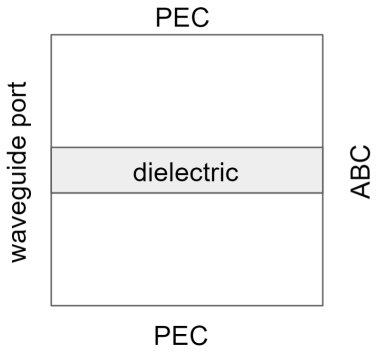


Figure 46: Domain of 2D Dielectric slab waveguide

dielectric, we set the relative permittivity $\epsilon_r = 2$. That is,

$$\epsilon_r = \begin{cases} 2 & \text{in dielectric slab,} \\ 1 & \text{otherwise.} \end{cases}$$

All the other settings are kept the same to the previous example.

10.3.1 Case setup

For the sake of simplicity, we only show the parts of code that are different than the previous example. The main difference is the spatially dependent permittivity. First, we will need to compute the eigenfunctions on the left boundary.

```
# helper function for computing laplacian eigen values
def Laplacian_1D_eig(a, b, N, eps=lambda x: np.ones_like(x), k=3):
  n = N - 2
  h = (b-a)/(N-1)

  L = diags([1, -2, 1], [-1, 0, 1], shape=(n, n))
  L = -L/h**2

  x = np.linspace(a, b, num=N)
  M = diags([eps(x[1:-1])], [0])

  eigvals, eigvecs = eigsh(L, k=k, M=M, which="SM")
  eigvecs = np.vstack((np.zeros((1,k)), eigvecs, np.zeros((1,k))))
  norm_eigvecs = np.linalg.norm(eigvecs, axis=0)
  eigvecs /= norm_eigvecs
  return eigvals.astype(np.float32), eigvecs.astype(np.float32), x.astype(np.float32)
```

Listing 72: 1D eigensolver

For the geometry part, we will need to define the slab and corresponding permittivity function.

```
len_slab = 0.6
eps0 = 1.
eps1 = 2.
eps_numpy = lambda y: np.where(np.logical_and(y>(height-len_slab)/2, y<(height+len_slab)/2), eps1, eps0)
eps_sympy = sqrt(eps0+(Heaviside(y-(height-len_slab)/2)-Heaviside(y-(height+len_slab)/2))*(eps1-eps0))
```

Listing 73: Define the slab and corresponding permittivity function

There is a square root in `eps_sympy` because in the `HelmholtzEquation`, the wavenumber will be squared. Next, based on the permittivity function, we use the eigensolver to get the numerical waveguide port.

```
eigvals, eigvecs, yv = Laplacian_1D_eig(0, height, 1000, eps=eps_numpy, k=3)
yv = yv.reshape((-1, 1))
eigenmode = [1]
wave_number = 16.    # wave_number = freq/c
waveguide_port_invar_numpy = {'x': np.zeros_like(yv), 'y': yv}
waveguide_port_outvar_numpy = {'u': 10*eigvecs[:, 0:1]}
```

Listing 74: Get the waveguide port

Now, we can define the `TrainDomain`. The only difference here is the left boundary, which will be given by a `numpy` array. We show only the modified BC below:

```
Wall = BC.from_numpy(waveguide_port_invar_numpy,
                     waveguide_port_outvar_numpy,
                     batch_size=200,
                     lambda_numpy={'lambda_u': np.full_like(yv, 0.5)})
self.add(Wall, name="Waveguide_port")
```

Listing 75: Training domain for 2D Dielectric slab

In the `Solver`, we set the `k` as the product of wavenumber and permittivity function. Also, we update the frequency for the Fourier features to suit the problem.

```
class WaveguideSolver(Solver):
    train_domain = HemholtzTrain
    inference_domain = WaveguideInference
    arch = ModifiedFourierNetArch

    def __init__(self, **config):
        super(WaveguideSolver, self).__init__(**config)

        self.equations = (HelmholtzEquation(u='u', k=wave_number*eps_sympy, dim=2).make_node()
                          + GradNormal(T='u', dim=2, time=False).make_node())

        self.arch.frequencies = ('axis,diagonal', [i / 2. for i in range(int(wave_number*np.sqrt(eps1))*2+1)])
        self.arch.frequencies_params = ('axis,diagonal', [i / 2. for i in range(int(wave_number*np.sqrt(eps1))
    *2+1)])
        wave_net = self.arch.make_node(name='wave_net',
                                       inputs=['x', 'y'],
                                       outputs=['u'])
        self.nets = [wave_net]
```

Listing 76: Solver for 2D Dielectric slab

10.3.2 Results

The full code of this example can be found in `examples/waveguide/dielectric_slab_waveguide_2D_TMz.py`. We do the simulation with wavenumber equals 16 and 32, respectively. The results are shown in Figure 47.

10.4 Problem 3: 3D waveguide cavity

In this example, we show how to setup a 3D waveguide simulation in Modulus. Unlike the previous examples, we will use the features in Modulus to define the boundary condition. The geometry is $\Omega = [0, 2]^3$, as shown in Figure 48.

10.4.1 Problem setup

We will solve the 3D frequency domain Maxwell's equation for electronic field $\mathbf{E} = (E_x, E_y, E_z)$:

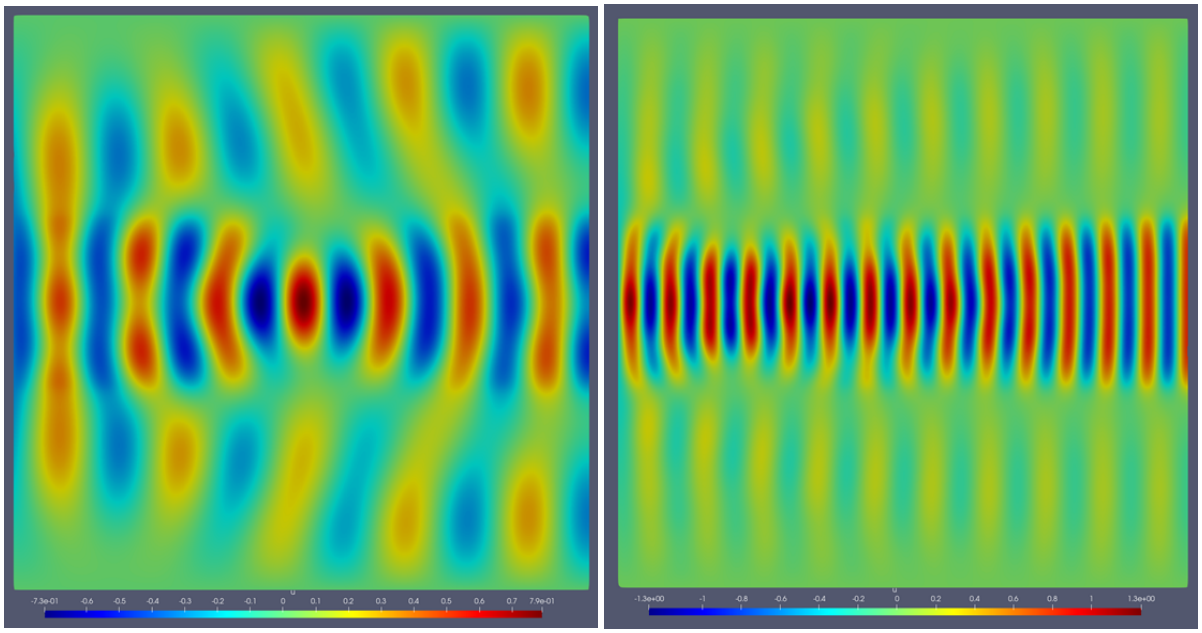
$$\nabla \times \nabla \times \mathbf{E} + \epsilon_r k^2 \mathbf{E} = 0,$$

where ϵ_r is the permittivity, and the k is the wavenumber. Note that, currently Modulus only support real permittivity and wavenumber. For the sake of simplicity, we assume the permeability $\mu_r = 1$. As before, waveguide port has been applied on the left. We apply absorbing boundary condition on the right side and PEC for the rest. In 3D, the absorbing boundary condition for real form reads

$$\mathbf{n} \times \nabla \times \mathbf{E} = 0,$$

while the PEC is

$$\mathbf{n} \times \mathbf{E} = 0.$$



(a) Modulus, wavenumber=16 (b) Modulus, wavenumber=32

Figure 47: Result of simulation. Eigenmode=1

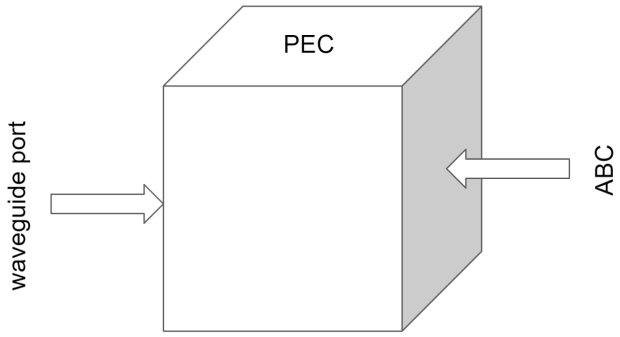


Figure 48: 3D waveguide geometry

10.4.2 Case setup

In this section, we will show how to use Modulus to setup the 3D frequency EM solver, especially for the boundary conditions.

We first import the necessary libraries.

```
from sympy import Symbol, pi, sin, Number, Eq
from sympy.logic.boolalg import And
from modulus.solver import Solver
from modulus.dataset import TrainDomain, InferenceDomain
from modulus.data import Inference
from modulus.sympy_utils.geometry_3d import Box
from modulus.PDES.Electromagnetic import PEC_3D, SommerfeldBC_real_3D, Maxwell_Freq_real_3D
from modulus.controller import ModulusController
from modulus.architecture.modified_fourier_net import ModifiedFourierNetArch
```

Listing 77: Import libraries

And, as before, we define `sympy` variables, geometry and waveguide function.

```
x, y, z = Symbol('x'), Symbol('y'), Symbol('z')
```

```
# params for domain
length = 2
height = 2
width = 2

eigenmode = [2]
wave_number = 32.    # wave_number = freq/c
waveguide_port = Number(0)
for k in eigenmode:
    waveguide_port += sin(k*pi*y/length)*sin(k*pi*z/height)

# define geometry
rec = Box((0, 0, 0),
          (width, length, height))
```

Listing 78: Preparations for the simulation

Then, we define the `TrainDomain` with PDEs, and all boundary conditions. The 3D Maxwell's equations has been implemented in `Maxwell_Freq_real_3D`, PEC has been implemented in `PEC_3D`, and absorbing boundary condition has been implemented in `SommerfeldBC_real_3D`. We may use these features directly to apply the corresponding constraints.

```
class HemholtzTrain(TrainDomain):
  def __init__(self, **config):
    super(HemholtzTrain, self).__init__()

    #walls
    Wall = rec.boundary_bc(outvar_sympy={'PEC_3D_x': 0., 'PEC_3D_y': 0., 'PEC_3D_z': 0.},
                           criteria=And(~Eq(x, 0), ~Eq(x, width)),
                           lambda_sympy={'lambda_PEC_3D_x': 100.,
                                         'lambda_PEC_3D_y': 100.,
                                         'lambda_PEC_3D_z': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="PEC")

    Wall = rec.boundary_bc(outvar_sympy={'uz': waveguide_port},
                           criteria=Eq(x, 0),
                           lambda_sympy={'lambda_uz': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="Waveguide_port")

    Wall = rec.boundary_bc(outvar_sympy={'SommerfeldBC_real_3D_x': 0.,
                                         'SommerfeldBC_real_3D_y': 0.,
                                         'SommerfeldBC_real_3D_z': 0.},
                           criteria=Eq(x, width),
                           lambda_sympy={'lambda_SommerfeldBC_real_3D_x': 10.,
                                         'lambda_SommerfeldBC_real_3D_y': 10.,
                                         'lambda_SommerfeldBC_real_3D_z': 10.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="ABC")

    # interior
    interior = rec.interior_bc(outvar_sympy={'Maxwell_Freq_real_3D_x': 0.,
                                             'Maxwell_Freq_real_3D_y': 0.,
                                             'Maxwell_Freq_real_3D_z': 0.},
                               bounds={x: (0, width), y: (0, length), z: (0, height)},
                               lambda_sympy={'lambda_Maxwell_Freq_real_3D_x': 1.0/wave_number**2,
                                             'lambda_Maxwell_Freq_real_3D_y': 1.0/wave_number**2,
                                             'lambda_Maxwell_Freq_real_3D_z': 1.0/wave_number**2},
                               fixed_var=False,
                               batch_size_per_area=1000)
    self.add(interior, name="Interior")
```

Listing 79: Define the train domain

Note that we are working in 3D, so the PDEs, PEC and absorbing boundaries have three output components.

Inference domain has been defined to check the result.

```
class WaveguideInference(InferenceDomain):
  def __init__(self, **config):
    super(WaveguideInference, self).__init__()
    interior_points = rec.sample_interior(100000, bounds={x: (0, width), y: (0, length), z: (0, height)})
    inf = Inference(interior_points, ['ux', 'uy', 'uz'])
```

```
    self.add(inf, name='Inf'+str(wave_number).zfill(4))
```

Listing 80: Define the inference domain

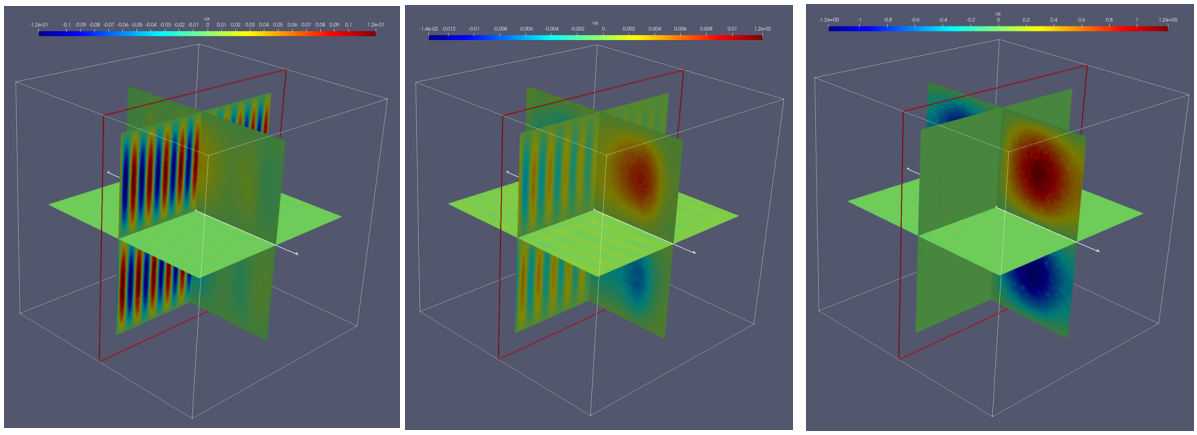
We finalize the code by defining the `Solver` and necessary hyperparameters for the neural network.

```
class WaveguideSolver(Solver):
  train_domain = HemholtzTrain
  inference_domain = WaveguideInference
  arch = ModifiedFourierNetArch

  def __init__(self, **config):
    super(WaveguideSolver, self).__init__(**config)

    self.equations = (Maxwell_Freq_real_3D(k=wave_number).make_node()
                      + PEC_3D().make_node()
                      + SommerfeldBC_real_3D().make_node())

    self.arch.frequencies = ('axis,diagonal', [i / 2. for i in range(int(wave_number)+1)])
    self.arch.frequencies_params = ('axis,diagonal', [i / 2. for i in range(int(wave_number)+1)])
    wave_net = self.arch.make_node(name='wave_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['ux', 'uy', 'uz'])
    self.nets = [wave_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_waveguide3D',
        'start_lr': 1e-3,
        'decay_steps': 10000,
        'max_steps': 500000,
        'layer_size': 512,
        'nr_layers': 6,
        'xla': True,
        })
```

Listing 81: Define neural network

10.4.3 Results

The full code of this example can be found in `examples/waveguide/waveguide3D.py`. We set the wavenumber equals 32 and use second eigenmode for y and z. The silces of the three components are shown in Figure 49.

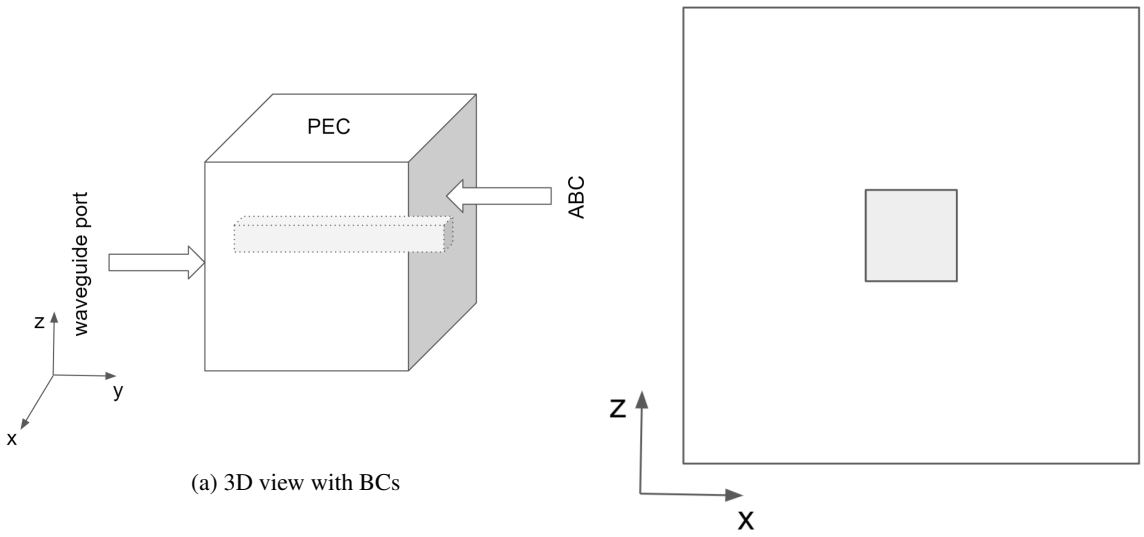


(a) 3D waveguide, E_x (b) 3D waveguide, E_y (c) 3D waveguide, E_z

Figure 49: Results

10.5 Problem 4: 3D Dielectric slab waveguide

In this example, we will show a 3D dielectric slab waveguide. In this case, we consider a unit cude $[0, 1]^3$ with a dielectric slab centered in the middle along y axis. The length of the slab is 0.2. Figure 50 shows the whole geometry and an xz cross-section that shows the dielectric slab.



(a) 3D view with BCs

(b) xz cross-sectional view

Figure 50: Geometry of 3D dielectric slab

The permittivity is defined as follows

$$\epsilon_r = \begin{cases} 1.5 & \text{in dielectric slab,} \\ 1 & \text{otherwise.} \end{cases}$$

10.5.1 Case setup

For the sake of simplicity, we only show the differences compared to the last example. The main difference for this simulation is that we will need to import the waveguide result from a `csv` file and then use that as the waveguide port boundary condition.

Let us first define the geometry and the `sympy` permittivity function:

```
# params for domain
length = 1
height = 1
width = 1

len_slab = 0.2
eps0 = 1.
eps1 = 1.5
eps_sympy = sqrt(eps0+(Heaviside(y+len_slab/2)-Heaviside(y-len_slab/2))
                 *(Heaviside(z+len_slab/2)-Heaviside(z-len_slab/2))*(eps1-eps0))

# define geometry
rec = Box((-width/2, -length/2, -height/2),
          (width/2, length/2, height/2))
```

Listing 82: Define the domain and permittivity function

Note that to define the piece-wise `sympy` functions, we should use `Heaviside` instead of `Piecewise` as the later cannot be complied in Modulus for the time being.

Next, we import the waveguide data.

```
mapping = {'x': 'x', 'y': 'y', **{'u'+str(k): 'u'+str(k) for k in range(6)}}
data_var = csv_to_dict('validation/2Dwaveguideport.csv', mapping)
wave_number = 32.    # wave_number = freq/c
waveguide_port_invar_numpy = {'x': np.zeros_like(data_var['x']), 'y': data_var['x'], 'z': data_var['y']}
waveguide_port_outvar_numpy = {'uz': data_var['u0']}
```

Listing 83: Import waveguide data

In validation/2Dwaveguideport.csv, there are six eigenmodes. You may try different modes to explore more interesting results.

Then, we define the TrainDomain. Please note that we use imported data as the waveguide port boundary condition.

```
class HemholtzTrain(TrainDomain):
    def __init__(self, **config):
        super(HemholtzTrain, self).__init__()

        #walls
        Wall = rec.boundary_bc(outvar_sympy={'PEC_3D_x': 0., 'PEC_3D_y': 0., 'PEC_3D_z': 0.},
                               criteria=And(~Eq(x, -width/2), ~Eq(x, width/2)),
                               lambda_sympy={'lambda_PEC_3D_x': 100.,
                                             'lambda_PEC_3D_y': 100.,
                                             'lambda_PEC_3D_z': 100.},
                               fixed_var=False,
                               batch_size_per_area=100)
        self.add(Wall, name="PEC")

        Wall = BC.from_numpy(waveguide_port_invar_numpy,
                             waveguide_port_outvar_numpy,
                             batch_size=200,
                             lambda_numpy={'lambda_uz': np.full_like(waveguide_port_invar_numpy['x'], 0.5)})
        self.add(Wall, name="Waveguide_port")

        Wall = rec.boundary_bc(outvar_sympy={'SommerfeldBC_real_3D_x': 0.,
                                             'SommerfeldBC_real_3D_y': 0.,
                                             'SommerfeldBC_real_3D_z': 0.},
                               criteria=Eq(x, width/2),
                               lambda_sympy={'lambda_SommerfeldBC_real_3D_x': 10.,
                                             'lambda_SommerfeldBC_real_3D_y': 10.,
                                             'lambda_SommerfeldBC_real_3D_z': 10.},
                               fixed_var=False,
                               batch_size_per_area=100)
        self.add(Wall, name="ABC")

        # interior
        interior = rec.interior_bc(outvar_sympy={'Maxwell_Freq_real_3D_x': 0.,
                                                 'Maxwell_Freq_real_3D_y': 0.,
                                                 'Maxwell_Freq_real_3D_z': 0.},
                                   bounds={x: (-width/2, width/2), y: (-length/2, length/2), z: (-height/2,
    height/2)},
                                   lambda_sympy={'lambda_Maxwell_Freq_real_3D_x': 1.0/wave_number**2,
                                                 'lambda_Maxwell_Freq_real_3D_y': 1.0/wave_number**2,
                                                 'lambda_Maxwell_Freq_real_3D_z': 1.0/wave_number**2},
                                   fixed_var=False,
                                   batch_size_per_area=1000)
        self.add(interior, name="Interior")
```

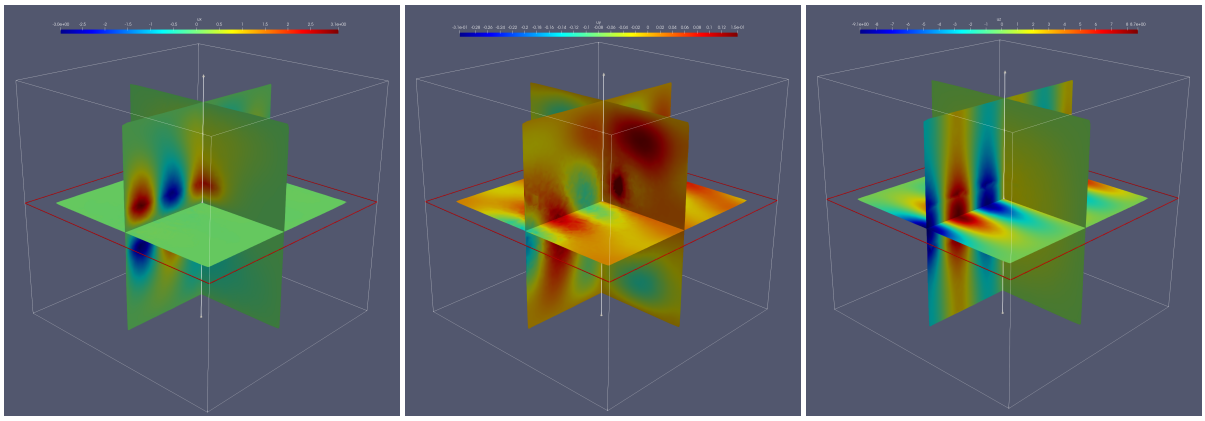
Listing 84: Define the TrainDomain

Finally, we define the InferenceDomain and Solver. These are same as the previous example except the bounds for the domain. The result is shown in Figure 51.

10.5.2 Results

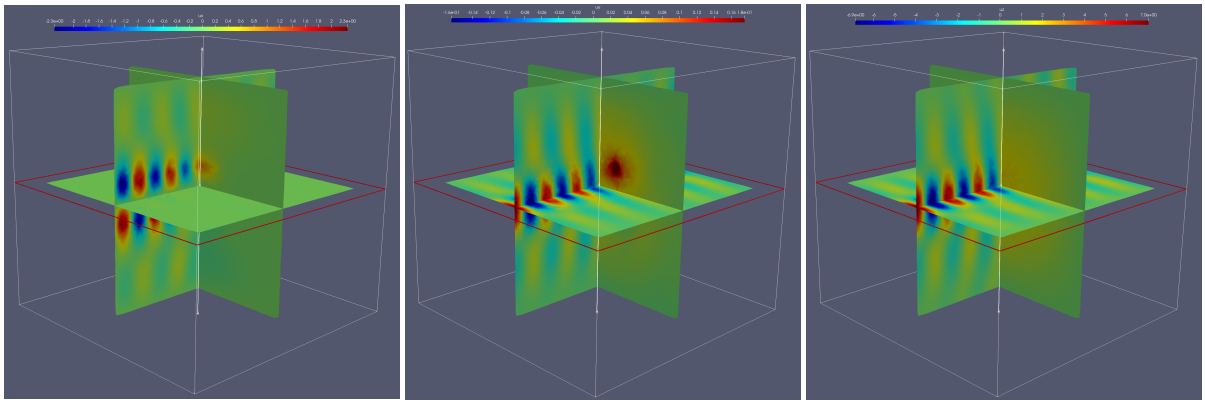
The full code of this example can be found in examples/waveguide/dielectric_slab_waveguide_3D.py. We do the simulation for different wavennumbers.

We also display the results of higher wavenumber 32 in Figure 52.



(a) 3D dielectric slab, E_x (b) 3D dielectric slab, E_y (c) 3D dielectric slab, E_z

Figure 51: Results for wavenumber 16



(a) 3D dielectric slab, E_x (b) 3D dielectric slab, E_y (c) 3D dielectric slab, E_z

Figure 52: Results for wavenumber 32

11 Linear Elasticity

11.1 Introduction

In this tutorial, we will walk you through the linear elasticity implementation in Modulus. Modulus offers the capability of solving the linear elasticity equations in the differential or variational form, allowing to solve a wide range of problems with a variety of boundary conditions. We present three examples in this chapter, namely the 3D bracket, the fuselage panel, and the plane displacement, to discuss the details of the linear elasticity in Modulus. Thus, in this tutorial, you would learn the following:

- How to solve linear elasticity equations using the differential and variational forms.
- How to solve linear elasticity problems for 3d and thin 2d structures (plane-stress).
- How to non-dimensionalize the elasticity equations.

11.2 Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you to refer Section 1.7 from the Theory chapter for more details on weak solutions of PDEs. Some of the boundary conditions are defined more elaborately in the tutorial 9 and we encourage you to refer that tutorial for more details.

Note: The linear elasticity equations in Modulus can be found in the source code directory `modulus/PDES/linear_elasticity.py`.

11.3 Linear Elasticity in the Differential Form

11.3.1 Linear elasticity equations in the displacement form

The displacement form of the (non-transient) linear elasticity equations, known as the Navier equations, is defined as

$$(\lambda + \mu)u_{j,ji} + \mu u_{i,jj} + f_i = 0, \tag{87}$$

where u_i is the displacement vector, f_i is the body force per unit volume, and λ, μ are the Lamé parameters defined as

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \tag{88}$$

$$\mu = \frac{E}{2(1+\nu)}. \tag{89}$$

Here, E, v are, respectively, the Young's modulus and Poisson's ratio.

11.3.2 Linear elasticity equations in the mixed form

In addition to the displacement formulation, linear elasticity can also be described by the mixed-variable formulation. In fact, based on our experiments and also the studies reported in [33], the mixed-variable formulation is easier for a neural network solver to learn, possibly due to the lower order differential terms. In the mixed-variable formulation, the equilibrium equations are defend as

$$\sigma_{ji,j} + f_i = 0, \tag{90}$$

where σ_{ij} is the Cauchy stress tensor. the stress-displacement equations are also defined as

$$\sigma_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}, \tag{91}$$

where δ_{ij} is the Kronecker delta function and ϵ_{ij} is the strain tensor that takes the following form

$$\epsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right). \tag{92}$$

11.3.3 Non-dimensionalized linear elasticity equations

It is often advantageous to work with the non-dimensionalized and normalized variation of the elasticity equations to improve the training convergence and accuracy. To this end, let us define our non-dimensionalized variables as

$$\hat{x}_i = \frac{x_i}{L}, \tag{93}$$

$$\hat{u}_i = \frac{u_i}{U}, \tag{94}$$

$$\hat{\lambda} = \frac{\lambda}{\mu_c}, \tag{95}$$

$$\hat{\mu} = \frac{\mu}{\mu_c} \tag{96}$$

Here, L is the characteristic length, U is the characteristic displacement, and μ_c is the non-dimensionalizing shear modulus. One can obtain the non-dimensionalized Navier and equilibrium equations by multiplying both sides of equations by $L^2/\mu_c U$, as follows

$$(\hat{\lambda} + \hat{\mu})\hat{u}_{j,ji} + \hat{\mu}\hat{u}_{i,jj} + \hat{f}_i = 0, \tag{97}$$

$$\hat{\sigma}_{ji,j} + \hat{f}_i = 0, \tag{98}$$

where the non-dimensionalized body force and stress tensor are

$$\hat{f}_i = \frac{L^2}{\mu_c U} f_i, \tag{99}$$

$$\hat{\sigma}_{ij} = \frac{L}{\mu_c U}\sigma_{ij}. \tag{100}$$

Similarly, the non-dimensionalized form of the stress-displacement equations are obtained by multiplying both sides of equations by $L/\mu_c U$, as follows

$$\hat{\sigma}_{ij} = \hat{\lambda}\hat{\epsilon}_{kk}\delta_{ij} + 2\hat{\mu}\hat{\epsilon}_{ij}, \tag{101}$$

$$\hat{\epsilon}_{ij} = \frac{1}{2}\left(\hat{u}_{i,j} + \hat{u}_{j,i}\right). \tag{102}$$

11.3.4 Plane stress equations

In a plane stress setting for thin structures, we assume that

$$\hat{\sigma}_{zz} = \hat{\sigma}_{xz} = \hat{\sigma}_{yz} = 0, \tag{103}$$

and therefore, the following relationship holds

$$\hat{\sigma}_{zz} = \hat{\lambda}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}} + \frac{\partial \hat{w}}{\partial \hat{z}}\right) + 2\hat{\mu}\frac{\partial \hat{w}}{\partial \hat{z}} = 0 \Rightarrow \frac{\partial \hat{w}}{\partial \hat{z}} = \frac{-\hat{\lambda}}{(\hat{\lambda} + 2\hat{\mu})}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}}\right). \tag{104}$$

Accordingly, the equations for $\hat{\sigma}_{xx}$ and $\hat{\sigma}_{yy}$ can be updated as follows

$$\hat{\sigma}_{xx} = \hat{\lambda}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}} + \frac{-\hat{\lambda}}{(\hat{\lambda} + 2\hat{\mu})}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}}\right)\right) + 2\hat{\mu}\frac{\partial \hat{u}}{\partial \hat{x}} \tag{105}$$

$$\hat{\sigma}_{yy} = \hat{\lambda}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}} + \frac{-\hat{\lambda}}{(\hat{\lambda} + 2\hat{\mu})}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}}\right)\right) + 2\hat{\mu}\frac{\partial \hat{v}}{\partial \hat{y}}. \tag{106}$$

11.3.5 Problem 1: Deflection of a bracket

In this first linear elasticity example, we consider a 3D bracket as shown in Figure 53. This example is partially adopted form the MATLAB PDE toolbox. The back face of this bracket is clamped, and a traction of 4×10^4 Pa is applied to the front face in the negative-z direction (this face is shown in red). The rest of the surface is considered as traction-free boundaries. We set $(E, \nu) = (100\,\text{GPa}, 0.3)$. We non-dimensionalize the linear elasticity equations by setting $L = 1\,\text{m}, U = 0.0001\,\text{m}, \mu_c = 0.01\mu$:

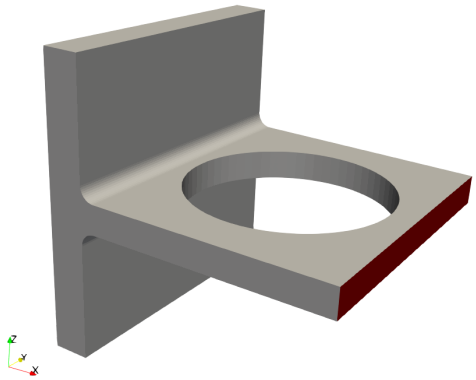


Figure 53: Geometry of the bracket. The back face of the bracket is clamped, and a shear stress is applied to the front face in the negatize-z direction.

```
# Parameters
nu                           = 0.3
E                            = 100e9
lambda_                      = nu*E/((1+nu)*(1-2*nu))
mu                           = E/(2*(1+nu))
mu_c                         = 0.01*mu
lambda_                      = lambda_/mu_c
mu                           = mu/mu_c
characteristic_length        = 1.
characteristic_displacement  = 1e-4
sigma_normalization          = characteristic_length/(characteristic_displacement*mu_c)
T                            = -4e4 * sigma_normalization
```

Listing 85: Non-dimensionalization for the bracket example.

As a rule of thumb, the characteristic length can be chosen in such a way to bound the largest dimension of the geometry to $(-1, 1)$. The characteristic displacement and μ_c can also be chosen such that the maximum displacement and the applied traction are close to 1 in order. Although the maximum displacement is not known a priori, we have observed that the convergence is not sensitive to the choice of the characteristic displacement as long as it is reasonably selected based on an initial guess for the approximate order of displacement. More information on non-dimensionalizing the PDEs can be found in Scaling of Differential Equations.

11.3.5.1 Case Setup and Results

We use the mixed-form of the linear elasticity equations here in this example, and therefore, we define the training domain as shown below. The complete python script for this problem can be found at `examples/bracket/bracket.py`.

```
class BracketTrain(TrainDomain):
  def __init__(self, **config):
    super(BracketTrain, self).__init__()

    backBC = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                             lambda_sympy={'lambda_u': 10,
                                           'lambda_v': 10,
                                           'lambda_w': 10},
                             batch_size_per_area=2000,
                             criteria= Eq(x, support_origin[0]))
    self.add(backBC, name="backBC")

    frontBC = geo.boundary_bc(outvar_sympy={'traction_x': 0,
                                            'traction_y': 0,
                                            'traction_z': T},
```

```
                                    lambda_sympy={'lambda_traction_x': 1,
                                                  'lambda_traction_y': 1,
                                                  'lambda_traction_z': 1},
                                    batch_size_per_area=2000,
                                    criteria=Eq(x, bracket_origin[0]+bracket_dim[0]))
    self.add(frontBC, name="frontBC")

    surfaceBC = geo.boundary_bc(outvar_sympy={'traction_x': 0,
                                              'traction_y': 0,
                                              'traction_z': 0},
                                lambda_sympy={'lambda_traction_x': 1,
                                              'lambda_traction_y': 1,
                                              'lambda_traction_z': 1},
                                batch_size_per_area=2000,
                                criteria= (x>support_origin[0]) &
                                          (x<bracket_origin[0]+bracket_dim[0]))
    self.add(surfaceBC, name="surfaceBC")

    interior = geo.interior_bc(outvar_sympy={'equilibrium_x': 0.,
                                             'equilibrium_y': 0.,
                                             'equilibrium_z': 0.,
                                             'stress_disp_xx': 0.,
                                             'stress_disp_yy': 0.,
                                             'stress_disp_zz': 0.,
                                             'stress_disp_xy': 0.,
                                             'stress_disp_xz': 0.,
                                             'stress_disp_yz': 0.},
                               lambda_sympy={'lambda_equilibrium_x': geo.sdf,
                                             'lambda_equilibrium_y': geo.sdf,
                                             'lambda_equilibrium_z': geo.sdf,
                                             'lambda_stress_disp_xx': geo.sdf,
                                             'lambda_stress_disp_yy': geo.sdf,
                                             'lambda_stress_disp_zz': geo.sdf,
                                             'lambda_stress_disp_xy': geo.sdf,
                                             'lambda_stress_disp_xz': geo.sdf,
                                             'lambda_stress_disp_yz': geo.sdf},
                               bounds={x: bounds_support_x,
                                       y: bounds_support_y,
                                       z: bounds_support_z},
                               batch_size_per_area=16000)
    self.add(interior, name="Interior_support")

    interior = geo.interior_bc(outvar_sympy={'equilibrium_x': 0.,
                                             'equilibrium_y': 0.,
                                             'equilibrium_z': 0.,
                                             'stress_disp_xx': 0.,
                                             'stress_disp_yy': 0.,
                                             'stress_disp_zz': 0.,
                                             'stress_disp_xy': 0.,
                                             'stress_disp_xz': 0.,
                                             'stress_disp_yz': 0.},
                               lambda_sympy={'lambda_equilibrium_x': geo.sdf,
                                             'lambda_equilibrium_y': geo.sdf,
                                             'lambda_equilibrium_z': geo.sdf,
                                             'lambda_stress_disp_xx': geo.sdf,
                                             'lambda_stress_disp_yy': geo.sdf,
                                             'lambda_stress_disp_zz': geo.sdf,
                                             'lambda_stress_disp_xy': geo.sdf,
                                             'lambda_stress_disp_xz': geo.sdf,
                                             'lambda_stress_disp_yz': geo.sdf},
                               bounds={x: bounds_bracket_x,
                                       y: bounds_bracket_y,
                                       z: bounds_bracket_z},
                               batch_size_per_area=16000)
    self.add(interior, name="Interior_bracket")
```

Listing 86: Training domain for the bracket example.

Note that the training domain consists of two different sets of interior points (i.e., `Interior_support` and `Interior_bracket`). This is done only to generate the interior points more efficiently. We use two separate neural networks for displacement and stresses, as follows

```
def __init__(self, **config):
  super(BracketSolver, self).__init__(**config)
  self.equations = (LinearElasticity(lambda_=lambda_, mu=mu, dim=3).make_node())
  elasticity_net_disp = self.arch.make_node(name='elasticity_net_disp',
                                            inputs=['x', 'y', 'z'],
                                            outputs=['u', 'v', 'w'])
  elasticity_net_stress = self.arch.make_node(name='elasticity_net_stress',
```

```
                                        inputs=['x', 'y', 'z'],
                                        outputs=['sigma_xx', 'sigma_yy',
                                                 'sigma_zz', 'sigma_xy',
                                                 'sigma_xz', 'sigma_yz'])
self.nets = [elasticity_net_disp, elasticity_net_stress]
```

Listing 87: Defining the PDEs and networks for the bracket example.

Figure 56 shows the Modulus results and also a comparison with a commercial solver results. The results of these two solvers show good agreement, with only a 8% difference in maximum bracket displacement.

11.3.6 Problem 2: Stress analysis for aircraft fuselage panel

In this example, we will use Modulus for the analysis of stress concentration in an aircraft fuselage panel. Depending on the altitude of the flying plane, the fuselage panel is exposed to different values of hoop stress that can cause accumulated damage to the panel over the time. Therefore, in cumulative damage modeling of aircraft fuselage for the purpose of design and maintenance of aircrafts, it is required to perform several stress simulations for different hoop stress values. Here, we consider a simplified aircraft fuselage panel as shown in Figure 55, and construct a parameterized model that, once trained, can predict the stress and displacement in the panel for different values of hoop stress.

11.3.6.1 Case Setup and Results The panel material is Aluminium 2024-T3, with $(E, \nu) = (73\,\text{GPa}, 0.33)$. We train a parameterized model with varying $\sigma_{hoop} \in (46, 56.5)$. Since the thickness of the panel is very small (i.e., 2 mm), we will solve the plane stress equations in this example. The python script for this problem can be found at `examples/fuselage_panel/panel_solver.py`. The plane stress form of the linear elasticity equations in Modulus can be called by using the `LinearElasticityPlaneStress` class:

```
self.equations = (LinearElasticityPlaneStress(lambda_=lambda_, mu=mu).make_node())
```

Listing 88: Using the plane stress equations in Modulus.

Figure 56 shows the Modulus results for panel displacements and stresses. For comparison, the commercial solver results are also presented in Figure 57. The Modulus and commercial solver results are in close agreement, with a difference of less than 5% in the maximum Von Mises stress.

11.4 Linear Elasticity in the Variational Form

11.4.1 Linear elasticity equations in the variational form

In addition to the differential form of the linear elasticity equations, Modulus also enables the use of variational form of these equations (Tutorial 9). In this section, we derive the linear elasticity equations in the variational form as implemented in Modulus. We will use the non-dimensionalized variables in this derivation. We start by forming the inner product of Equation 98 and a vector test function $v \in \mathbf{V}$, and integrating over the domain, as follows

$$\int_\Omega \hat{\sigma}_{ji,j} v_i d\mathbf{x} + \int_\Omega \hat{f}_i v_i d\mathbf{x} = 0. \tag{107}$$

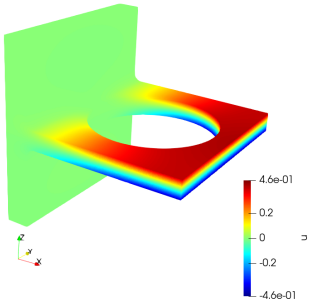
Using the integration by parts, we have

$$\int_{\partial\Omega} \hat{T}_i v_i d\mathbf{s} - \int_\Omega \hat{\sigma}_{ji} v_{j,i} d\mathbf{x} + \int_\Omega \hat{f}_i v_i d\mathbf{x} = 0, \tag{108}$$

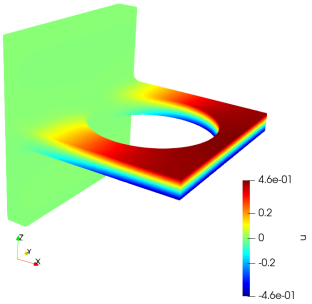
where T_i is the traction. Note that the first term is zero for the traction-free boundaries. Equation 108 is the variational form of the linear elasticity equations that is adopted in Modulus. The term $\hat{\sigma}_{ji}$ in this equation is computed using the stress-displacement relation in Equation 101.

11.4.2 Problem 3: Plane displacement

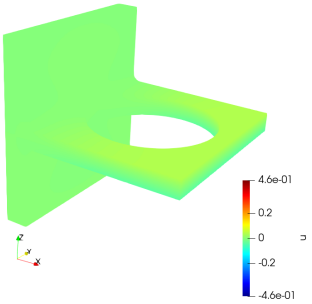
In this example, we solve the linear elasticity plane stress equations in the variational form. We consider a square plate that is clamped from one end and is under a displacement boundary condition on the other end, as shown in Figure 58. The rest of the boundaries are traction-free. The material properties are assumed to be $(E, \nu) = (10\,\text{MPa}, 0.2)$. This example is adopted from [33].



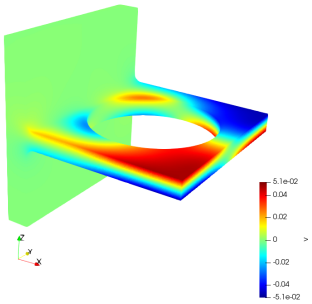
(a) u (Modulus)



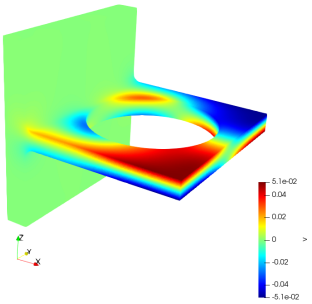
(b) u (Commercial solver)



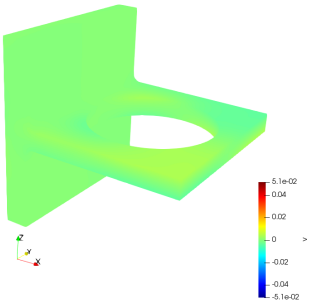
(c) u (Difference)



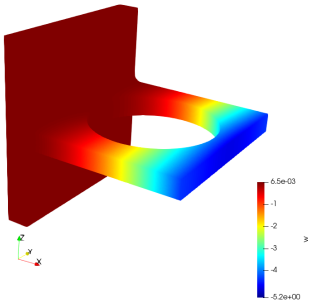
(d) v (Modulus)



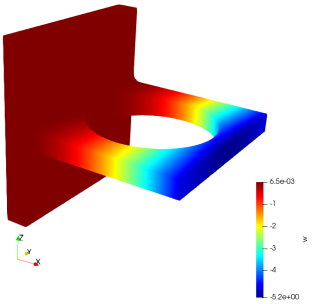
(e) v (Commercial solver)



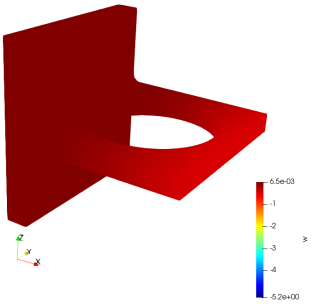
(f) v (Difference)



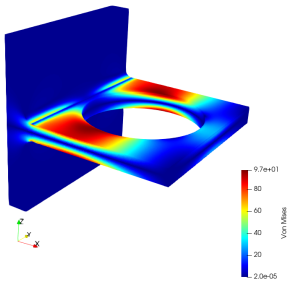
(g) w (Modulus)



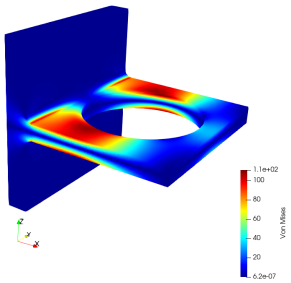
(h) w (Commercial Solver)



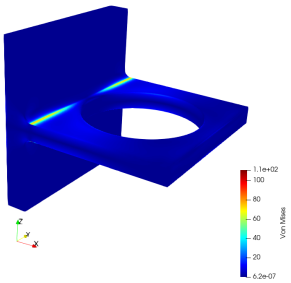
(i) w (Difference)



(j) σ_{VM} (Modulus)



(k) σ_{VM} (Commercial Solver)



(l) σ_{VM} (Difference)

Figure 54: Modulus linear elasticity results for the bracket deflection example.

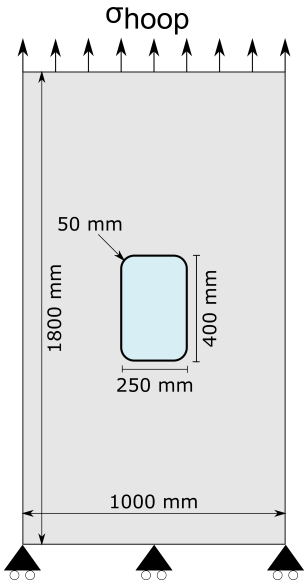
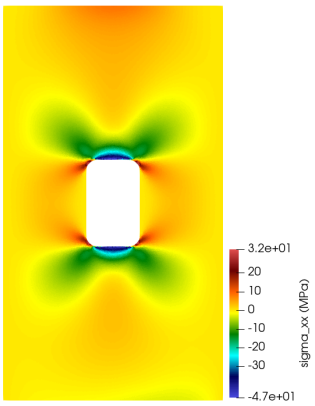
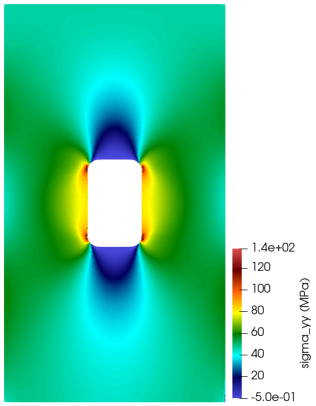


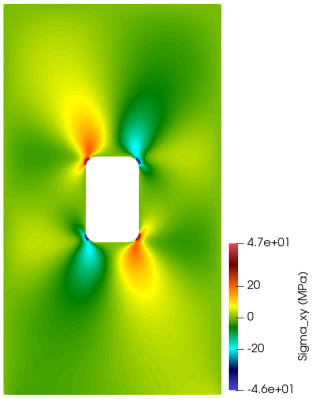
Figure 55: Geometry and boundary conditions of the simplified aircraft fuselage panel.



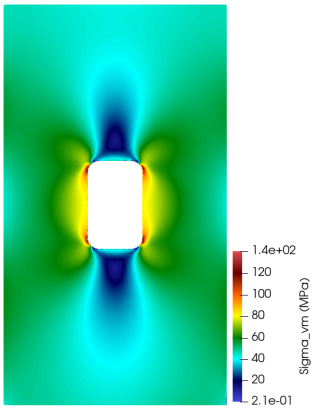
(a) σ_{xx}



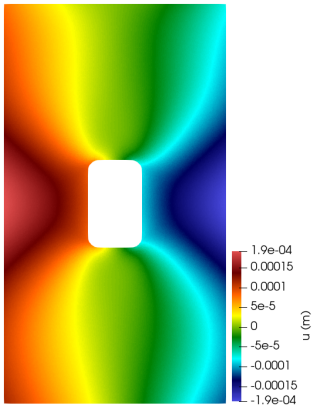
(b) σ_{yy}



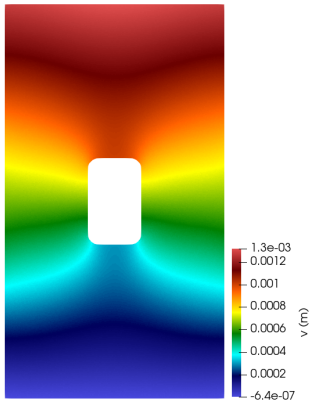
(c) σ_{xy}



(d) Von Mises stress

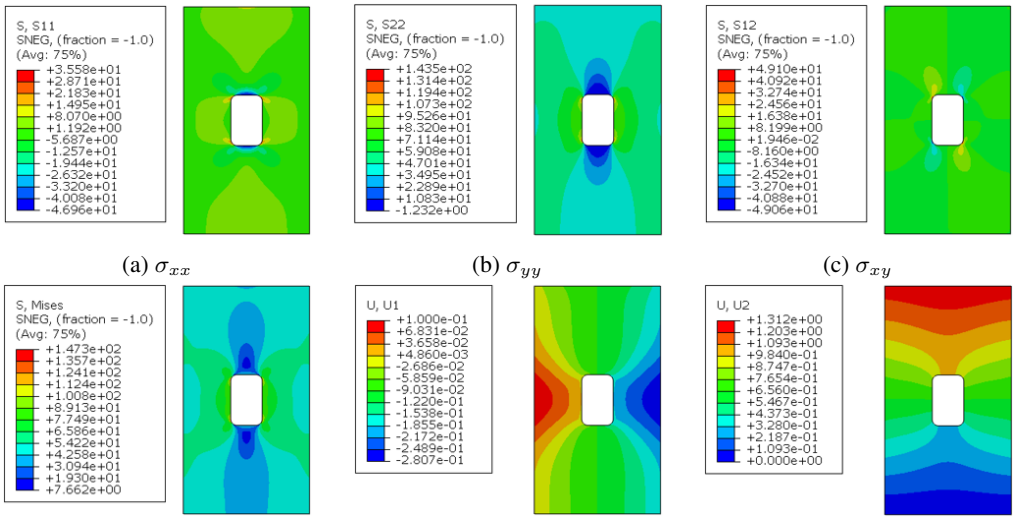


(e) u



(f) v

Figure 56: Modulus linear elasticity results for the aircraft fuselage panel example with parameterized hoop stress. The results are for $\sigma_{hoop} = 46$.



(a) σ_{xx} (b) σ_{yy} (c) σ_{xy}

(d) Von Mises stress (e) u (f) v

Figure 57: commercial solver linear elasticity results for the aircraft fuselage panel example with $\sigma_{hoop} = 46$.

11.4.2.1 Case Setup and Results To solve this problem in the variational form, we enforce the displacement boundary conditions in the differential form, and also generate interior and boundary points to be used in evaluation of the integrals in Equation 108. The python script for this problem can be found at `examples/plane_displacement/plane_displacement.py`.

```
class PlaneDisplacementTrain(TrainDomain):
  def __init__(self, **config):
    super(PlaneDisplacementTrain, self).__init__()
    bottomBC = geo.boundary_bc(outvar_sympy={'u': 0., 'v': 0.},
                               lambda_sympy={'lambda_u': 10,
                                             'lambda_v': 10},
                               batch_size_per_area=500,
                               criteria=Eq(y, domain_origin[1]),
                               batch_per_epoch=5000)
    self.add(bottomBC, name="bottomBC_differential")

    bottomBC = geo.boundary_bc(outvar_sympy={},
                               batch_size_per_area=500,
                               criteria=Eq(y, domain_origin[1]),
                               fixed_var=False,
                               quasirandom=True)
    self.add(bottomBC, name="bottomBC")

    topBC = geo.boundary_bc(outvar_sympy={'u': 0., 'v': 0.1},
                            lambda_sympy={'lambda_u': 10,
                                          'lambda_v': 10},
                            batch_size_per_area=500,
                            criteria=Eq(y, domain_origin[1]+ domain_dim[1])
                                     & (x<= domain_origin[0]+ domain_dim[0]/2.),
                            batch_per_epoch=5000)
    self.add(topBC, name="topBC_differential")

    topBC = geo.boundary_bc(outvar_sympy={},
                            batch_size_per_area=500,
                            criteria=Eq(y, domain_origin[1]+ domain_dim[1])
                                     & (x<= domain_origin[0]+ domain_dim[0]/2.),
                            fixed_var=False,
                            quasirandom=True)
    self.add(topBC, name="topBC")

    interior = geo.interior_bc(outvar_sympy={},
                               batch_size_per_area=10000,
                               bounds={x: bounds_x, y: bounds_y},
                               fixed_var=False,
                               quasirandom=True)
    self.add(interior, name="Interior")
```

Listing 89: Training domain for the plane displacement example.

Note that for the training points that are used in the variational form, we set the `outvar_sympy` to be an empty dictionary.

So far, we have only included the displacement boundary conditions in our loss function. We will define a custom loss function that includes the variational loss, and add that to the overall loss. In the remainder of this subsection, we will discuss how to generate this variational loss. We will first compute the neural network solution at the interior and boundary points, and then compute the required gradients:

```
def custom_loss(self, domain_invar, pred_domain_outvar, true_domain_outvar):
  # get points on the interior
  x_interior = domain_invar['Interior']['x']
  y_interior = domain_invar['Interior']['y']
  area_interior = domain_invar['Interior']['area']

  # compute solution for the interior
  u_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['u']
  u_x_interior = tf.gradients(u_interior, x_interior)[0]
  u_y_interior = tf.gradients(u_interior, y_interior)[0]
  v_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['v']
  v_x_interior = tf.gradients(v_interior, x_interior)[0]
  v_y_interior = tf.gradients(v_interior, y_interior)[0]

  # get points on the boundary
  x_bottom_dir = domain_invar['bottomBC']['x']
  y_bottom_dir = domain_invar['bottomBC']['y']
  normal_x_bottom_dir = domain_invar['bottomBC']['normal_x']
  normal_y_bottom_dir = domain_invar['bottomBC']['normal_y']
  area_bottom_dir = domain_invar['bottomBC']['area']

  x_top_dir = domain_invar['topBC']['x']
  y_top_dir = domain_invar['topBC']['y']
  normal_x_top_dir = domain_invar['topBC']['normal_x']
  normal_y_top_dir = domain_invar['topBC']['normal_y']
  area_top_dir = domain_invar['topBC']['area']

  # compute solution for the boundary
  u_bottom_dir = self.nets[0].evaluate({'x': x_bottom_dir, 'y': y_bottom_dir})['u']
  u_x_bottom_dir = tf.gradients(u_bottom_dir, x_bottom_dir)[0]
  u_y_bottom_dir = tf.gradients(u_bottom_dir, y_bottom_dir)[0]
  v_bottom_dir = self.nets[0].evaluate({'x': x_bottom_dir, 'y': y_bottom_dir})['v']
  v_x_bottom_dir = tf.gradients(v_bottom_dir, x_bottom_dir)[0]
  v_y_bottom_dir = tf.gradients(v_bottom_dir, y_bottom_dir)[0]

  u_top_dir = self.nets[0].evaluate({'x': x_top_dir, 'y': y_top_dir})['u']
  u_x_top_dir = tf.gradients(u_top_dir, x_top_dir)[0]
  u_y_top_dir = tf.gradients(u_top_dir, y_top_dir)[0]
  v_top_dir = self.nets[0].evaluate({'x': x_top_dir, 'y': y_top_dir})['v']
  v_x_top_dir = tf.gradients(v_top_dir, x_top_dir)[0]
  v_y_top_dir = tf.gradients(v_top_dir, y_top_dir)[0]
```

Listing 90: Computing the neural network solution and its gradients at the interior and boundary points for the plane displacement example.

In the next step, we define our test functions, and compute the test functions and their required gradients at the interior and boundary points:

```
  # test functions
  test_fn = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
                                         Trig_test: [k for k in range(10)]},
                                         box =[[domain_origin[0], domain_origin[1]],
                                               [domain_origin[0]+domain_dim[0],
                                                domain_origin[1]+domain_dim[1]]],
                                         diff_list=['grad'])

  v = Vector_Test(test_fn,test_fn,mix=0.02)
  vx_x_interior, vy_x_interior = v.eval_test('vx', x_interior, y_interior)
  vx_y_interior, vy_y_interior = v.eval_test('vy', x_interior, y_interior)
  vx_bottom_dir, vy_bottom_dir = v.eval_test('v', x_bottom_dir, y_bottom_dir)
  vx_top_dir, vy_top_dir = v.eval_test('v', x_top_dir, y_top_dir)
```

Listing 91: Defining the test functions for the plane displacement example.

Here, we construct a set of test functions consisting of Legendre polynomials and trigonometric functions, and randomly subsample 2% of these test functions. Note that we only compute the terms that appear in the variational loss in

Equation 108. For instance, it is not necessary to compute the derivative of the test functions with respect to input coordinates for the boundary points.

The next step is to compute the stress terms according to the plane stress equations in Equations 105, 106, and also the traction terms:

```
w_z_interior          = -lambda_/(lambda_+2*mu) * (u_x_interior+v_y_interior)
sigma_xx_interior     = lambda_ * (u_x_interior+v_y_interior+w_z_interior)
                        + 2*mu * u_x_interior
sigma_yy_interior     = lambda_ * (u_x_interior+v_y_interior+w_z_interior)
                        + 2*mu * v_y_interior
sigma_xy_interior     = mu * (u_y_interior+v_x_interior)

w_z_bottom_dir        = -lambda_/(lambda_+2*mu) * (u_x_bottom_dir+v_y_bottom_dir)
sigma_xx_bottom_dir   = lambda_ * (u_x_bottom_dir+v_y_bottom_dir+w_z_bottom_dir)
                        + 2*mu * u_x_bottom_dir
sigma_yy_bottom_dir   = lambda_ * (u_x_bottom_dir+v_y_bottom_dir+w_z_bottom_dir)
                        + 2*mu * v_y_bottom_dir
sigma_xy_bottom_dir   = mu * (u_y_bottom_dir+v_x_bottom_dir)

w_z_top_dir           = -lambda_/(lambda_+2*mu) * (u_x_top_dir+v_y_top_dir)
sigma_xx_top_dir      = lambda_ * (u_x_top_dir+v_y_top_dir+w_z_top_dir)
                        + 2*mu * u_x_top_dir
sigma_yy_top_dir      = lambda_ * (u_x_top_dir+v_y_top_dir+w_z_top_dir)
                        + 2*mu * v_y_top_dir
sigma_xy_top_dir      = mu * (u_y_top_dir+v_x_top_dir)

traction_x_bottom_dir = sigma_xx_bottom_dir*normal_x_bottom_dir
                        +sigma_xy_bottom_dir*normal_y_bottom_dir
traction_y_bottom_dir = sigma_xy_bottom_dir*normal_x_bottom_dir
                        +sigma_yy_bottom_dir*normal_y_bottom_dir
traction_x_top_dir    = sigma_xx_top_dir*normal_x_top_dir
                        +sigma_xy_top_dir*normal_y_top_dir
traction_y_top_dir    = sigma_xy_top_dir*normal_x_top_dir
                        +sigma_yy_top_dir*normal_y_top_dir
```

Listing 92: Computing the stress and traction terms for the plane displacement example.

Finally, following the Equation 108, we define our variational interior and boundary integral terms, formulate the variational loss, and add that to the overall loss, as follows

```
interior_loss = tensor_int(area_interior, sigma_xx_interior*vx_x_interior
                                          +sigma_yy_interior*vy_y_interior
                                          +sigma_xy_interior*(vx_y_interior
                                                              +vy_x_interior))
boundary_loss1 = tensor_int(area_bottom_dir, traction_x_bottom_dir*vx_bottom_dir
                                             +traction_y_bottom_dir*vy_bottom_dir)
boundary_loss2 = tensor_int(area_top_dir, traction_x_top_dir*vx_top_dir
                                         +traction_y_top_dir*vy_top_dir)

# make variational loss function
loss = Variables()
loss['loss_variational'] = tf.reduce_sum(tf.square(interior_loss - boundary_loss1 - boundary_loss2))
tf.summary.scalar('loss_variational', loss['loss_variational'])
return loss
```

Listing 93: Constructing the variational loss for the plance displacement example.

Figure 58 shows the Modulus results for this plane displacement example. The results are in good agreement with the FEM results reported in [33], verifying the accuracy of the Modulus results in the variational form.

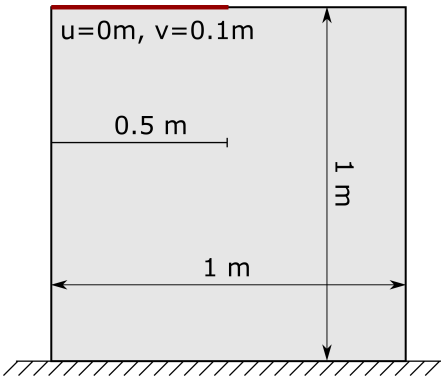
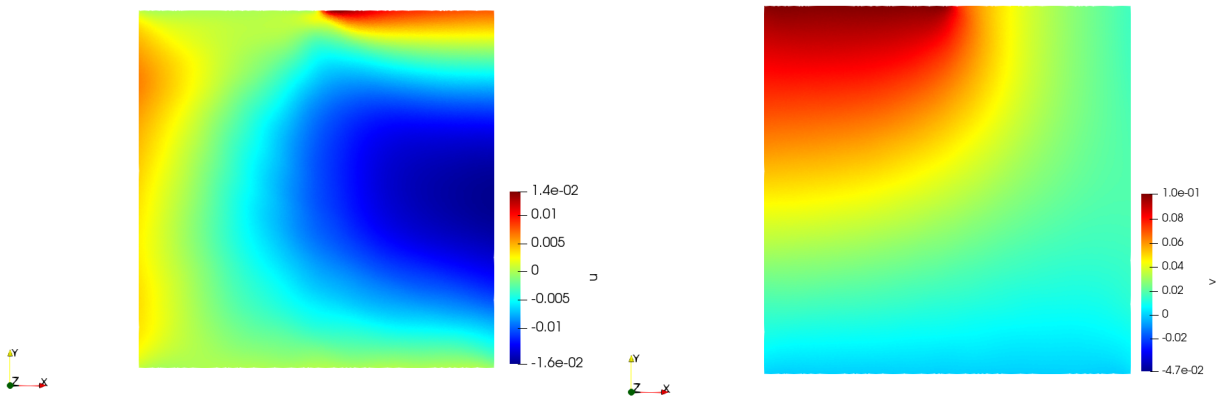


Figure 58: Geometry and boundary conditions of the plane displacement example. This example is adopted from [33].



(a) u

(b) v

Figure 59: Modulus results for the plane displacement example.

12 Tuning Neural Network Hyperparameters & Using Modulus' Advanced Features

12.1 Introduction

In this tutorial, we will briefly cover the sections of Modulus library which you can refer to make any modifications to the Neural Network architecture, implement some of the advanced Modulus features as well as perform hyper-parameter tuning.

Note: Unlike other tutorials, we will not cover the details of the code implementation. This tutorial will serve as a one-stop guide for all the important parameters that are at your disposal for hyperparameter tuning from an application perspective. However, for any details on the theory or the code implementation, we encourage you to visit the tutorial 1 and the source code documentation respectively.

Let's get started.

Most of the changes like modifying the number of hidden layers, activation function, learning rate parameters, etc. can be modified by `update_defaults` class method while creating the `Solver` class (Base class can be found at `modulus/solver.py`) or by passing flags while executing the command. We have already used the `update_defaults` method for specifying the network directory to write, max steps for the problem, etc. in previous tutorials.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_hemholtz',
        'rec_results': True,
        'rec_results_freq': 1000,
        'start_lr': 3e-4,
        'decay_steps': 20000,
        'max_steps': 40000
        })
```

Listing 94: Update defaults

The same can be achieved by the following command while executing the script.

```
python helmholtz.py --rec_results=True --rec_results_freq=1000 --start_lr=3e-4 --
    decay_steps=20000
```

To see the available options for hyper-parameter tuning, one can refer to the source code documentation or simply run the python script with `-h` flag.

```
python filename.py -h
```

The list of options should be displayed in the command window. An example is shown below:

```
usage: ldc_2d.py [-h] [--run_mode {solve,eval,plot_data}]
                 [--network_dir NETWORK_DIR]
                 [--initialize_network_dir INITIALIZE_NETWORK_DIR]
                 [--added_config_dir ADDED_CONFIG_DIR]
                 [--rec_results REC_RESULTS]
                 [--rec_results_freq REC_RESULTS_FREQ] [--max_steps MAX_STEPS]
                 [--save_filetypes SAVE_FILETYPES] [--xla XLA]
                 [--inner_norm INNER_NORM] [--outer_norm OUTER_NORM]
                 [--activation_fn ACTIVATION_FN] [--layer_size LAYER_SIZE]
                 [--nr_layers NR_LAYERS] [--skip_connections SKIP_CONNECTIONS]
                 [--weight_norm WEIGHT_NORM] [--start_lr START_LR]
                 [--end_lr END_LR] [--decay_steps DECAY_STEPS]
                 [--decay_rate DECAY_RATE] [--beta1 BETA1] [--beta2 BETA2]
                 [--epsilon EPSILON] [--amp AMP]

optional arguments:
  -h, --help            show this help message and exit

Controller Details:
  --run_mode {solve,eval,tkinter,plot_data}
                        all modes

Solver Configs:
  --network_dir NETWORK_DIR
                        where to save neural network solver
  --initialize_network_dir INITIALIZE_NETWORK_DIR
                        restore weights from this dir. This can be used to
                        solve one-way coupled systems sequentially for example
  --added_config_dir ADDED_CONFIG_DIR
```

```
                        configs to add onto network_dir seperated by comma
  --rec_results REC_RESULTS
                        record results while training
  --rec_results_freq REC_RESULTS_FREQ
                        steps till recording results
  --max_steps MAX_STEPS
                        max train steps
  --save_filetypes SAVE_FILETYPES
                        file types to save results too [vtk, csv, np]
  --xla XLA             use the XLA graph compiler?
  --inner_norm INNER_NORM
                        what norm to use to compute the loss for a single
                        field? (L1/L2)
  --outer_norm OUTER_NORM
                        what norm to use to compute the loss across fields?
                        (L1/L2)

Arch Configs:
  --activation_fn ACTIVATION_FN
                        nonlinearity for network
  --layer_size LAYER_SIZE
                        hidden layer size
  --nr_layers NR_LAYERS
                        nr layers in net
  --skip_connections SKIP_CONNECTIONS
                        residual skip connections
  --weight_norm WEIGHT_NORM
                        residual skip connections

LR Configs:
  --start_lr START_LR   start at this value
  --end_lr END_LR       decay to this value
  --decay_steps DECAY_STEPS
                        how many steps to take for decay
  --decay_rate DECAY_RATE
                        decay rate

Optimizer Configs:
  --beta1 BETA1         beta 1
  --beta2 BETA2         beta 2
  --epsilon EPSILON     epsilon
  --amp AMP             use Automatic Mixed Precision (AMP)? (0/1)
```

12.2 Network Architecture

The different architectures implemented in Modulus can be found in `modulus.architecture` module. Currently, Modulus has eight architectures for you to choose from: (1) fully-connected neural network (`fully_connected.py`), (2) Fourier network (`fourier_net.py`), (3) SiReN (`siren.py`), (4) Combination of Perdikaris architecture [10] and Fourier Networks (`modified_fourier_net.py`), (5) Combination of highway networks [11] and Fourier networks (`highway_fourier_net.py`), (6) DGM architecture (`dgm.py`), (7) multiplicative filter networks (`multiplicative_filter_net.py`), and (8) radial basis network (`radial_basis.py`).

A network is imported in from the `architecture` module and can be specified while instantiating the `Solver` class. An example is shown below.

```
from modulus.architecture.radial_basis import RadialBasisArch

# Define neural network
class ChipSolver(Solver):
  train_domain     = LDCTrain
  val_domain       = LDCVal
  arch             = RadialBasisArch
```

Listing 95: Selecting the architecture

The parameters to tune for the `FullyConnectedArch` are, `activation_fn`, `layer_size`, `nr_layers`, `skip_connections` and `weight_norm`. They can be modified in `update_defaults` as shown below.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'layer_size': 512,
        'nr_layer': 8,
        'skip_connections': False
        })
```

Listing 96: Changing the layers and layer size

For more information on the default values for these parameters, please refer to individual architecture files or the source code documentation.

12.3 Activation Functions

The complete list of activation functions available in Modulus can be found in the source code location `modulus/tf_utils/activation_functions.py`. The default activation function in Modulus is the swish but we also have implementations of activation functions like sin, tanh, etc.

The activation function can be changed as shown below.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'activation_fn': 'tanh'
        })
```

Listing 97: Changing activation functions

By default, Modulus does not use adaptive activation functions (described in Section 1.6.2.4). However, it is easy to turn on this feature, as shown below.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'adaptive_activations': True
        })
```

Listing 98: Using adaptive activations

12.4 Learning Rate Schedule

The different learning rate schedules available in Modulus can be found in the source code location `modulus/learning_rate.py`. The default is an exponential decay, but you can choose among exponential decay, polynomial decay, exponential decay with warm-up or cyclic schedules for the learning rates.

An example to use polynomial decay is shown below.

```
from modulus.learning_rate import LR, PolynomialDecayLR

# Define neural network
class ChipSolver(Solver):
  train_domain     = LDCTrain
  val_domain       = LDCVal
  lr               = PolynomialDecayLR

  def update_defaults(cls, defaults):
    defaults.update({
        'start_lr': 3e-04,
        'decay_steps': 2000,
        'power': 0.5
        })
```

Listing 99: Changing learning rate schedule

For more information on the default values for these parameters, please refer to the source code documentation.

12.5 Optimizers

Currently, Modulus only has the Adam optimizer implemented. You can modify the optimizer settings or add your own implementation of optimizer in the source code at `modulus/optimizer.py` and source it appropriately.

For more information on the default values for the optimizer parameters, please refer to the source code documentation.

By default, Modulus does not use the global or local learning rate annealing (as described in Section 1.6.2.1). A sample for using the global learning rate annealing for the lid-driven cavity example (as described in Section 2) is shown below.

```
# import Modulus library
from modulus.optimizer import AdamOptimizerAnnealing

class LDCSolver(Solver):
  optimizer = AdamOptimizerAnnealing

  def __init__(self, **config):
    super(LDCSolver, self).__init__(**config)
    self.optimizer.grouping = [{'continuity'},
                               {'momentum_x','momentum_y'},
                               {'u','v','p'}]
  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'local_annealing': False
    })
```

Listing 100: Using the global learning rate annealing for the lid-driven cavity example.

In order to use the local learning rate annealing instead, you can simply set the `'local_annealing'` to `True`. In `self.optimizer.grouping`, we specify which loss terms will be grouped together. Please note that all of the defined loss terms should be included here.

12.6 Gradient Aggregation

By default, Modulus does not use gradient aggregation. The following shows how to activate gradient aggregation for the annular ring example by using the `AdamOptimizerGradAgg` class.

```
from modulus.optimizer import AdamOptimizerGradAgg
grad_agg_freq = 2 # number of gradient aggregations

class AnnularRingTrain(TrainDomain):
  def __init__(self, **config):
    super(AnnularRingTrain, self).__init__()
    # inlet
    inlet_sympy = parabola(y, inter_1=channel_width[0], inter_2=channel_width[1], height=inlet_vel)
    inlet = geo.boundary_bc(outvar_sympy={'u': inlet_sympy, 'v': 0},
                            batch_size_per_area=32,
                            batch_per_epoch = 1000*grad_agg_freq,
                            criteria=Eq(x, channel_length[0]))
    self.add(inlet, name="Inlet")

    # outlet
    outlet = geo.boundary_bc(outvar_sympy={'p': 0},
                             batch_size_per_area=32,
                             batch_per_epoch = 1000*grad_agg_freq,
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="Outlet")

    # noslip
    noslip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                             batch_size_per_area=32,
                             batch_per_epoch = 1000*grad_agg_freq,
                             criteria=(x>channel_length[0])&(x<channel_length[1]))
    self.add(noslip, name="NoSlip")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0},
                               bounds={x: channel_length,
                                       y: (-outer_cylinder_radius, outer_cylinder_radius)},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                                             'lambda_momentum_x': geo.sdf,
                                             'lambda_momentum_y': geo.sdf},
                               batch_size_per_area=128,
                               batch_per_epoch = 1000*grad_agg_freq)
    self.add(interior, name="Interior")

    # make integral continuity
    outlet = geo.boundary_bc(outvar_sympy={'integral_continuity': 2},
                             batch_size_per_area=128,
                             batch_per_epoch = 1000*grad_agg_freq
                             lambda_sympy={'lambda_integral_continuity': 0.1},
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="OutletContinuity")

class ChipSolver(Solver):
```

```
  train_domain = AnnularRingTrain
  val_domain = AnnularRingVal
  inference_domain = AnnularRingInference
  monitor_domain = AnnularRingMonitor
  optimizer = AdamOptimizerGradAgg

  def __init__(self, **config):
    super(ChipSolver, self).__init__(**config)
    self.equations = (NavierStokes(nu=0.01, rho=1, dim=2, time=False).make_node()
                      + IntegralContinuity(dim=2).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    self.nets = [flow_net]
    self.save_network_freq = 1000*grad_agg_freq
    self.print_stats_freq = 100*grad_agg_freq
    self.tf_summary_freq = 500*grad_agg_freq

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_annular_ring_grad_agg',
        'decay_steps': 2000*grad_agg_freq,
        'max_steps': 200000*grad_agg_freq,
        'grad_agg_freq': grad_agg_freq
        })
```

Listing 101: Using gradient aggregation.

Here, `grad_agg_freq` represents the number of gradient aggregations per model parameter update. Effectively, setting `grad_agg_freq=2` will double the batch size per iteration, compared to a case with no gradient aggregation. We also set the `batch_per_epoch` to `1000*grad_agg_freq` to scale the size of the fixed point cloud by the number of aggregations (the default value for `batch_per_epoch` is 1000). This will keep the ratio between the size of the fixed point cloud and the batch size the same as for the case with no gradient aggregation. Please note that when gradient aggregation is used, the training iteration counter is updated at every gradient aggregation step (and not only at each model parameter update). Therefore, to keep everything consistent with the case where no gradient aggregation is used, we will also scale `self.save_network_freq, self.print_stats_freq, self.tf_summary_freq, decay_steps, max_steps` by the number of gradient aggregations.

12.7 Importance Sampling

Here we demonstrate the importance sampling on the interior domain of the annular ring example. The script is available at `annular_ring/annular_ring_importance_sampling.py`. After a warmup phase that the training points are sampled uniformly, at every 100 iterations, we compute the velocity derivatives on a 1024×512 grid, and approximate the sampling probability proportional to the 2-norm of the velocity gradients on the interior using a nearest neighbor interpolation. The interior points are then sampled according to these sampling probabilities. As shown in Equation 19, a correction factor is multiplied by the loss at each training point to avoid bias in the loss estimation.

```
class AnnularRingTrain(TrainDomain):
  def __init__(self, **config):
    super(AnnularRingTrain, self).__init__(nr_threads=1)

    # inlet
    inlet_sympy = parabola(y, inter_1=channel_width[0], inter_2=channel_width[1], height=inlet_vel)
    inlet = geo.boundary_bc(outvar_sympy={'u': inlet_sympy, 'v': 0},
                            batch_size_per_area=32,
                            criteria=Eq(x, channel_length[0]))
    self.add(inlet, name="Inlet")

    # outlet
    outlet = geo.boundary_bc(outvar_sympy={'p': 0},
                             batch_size_per_area=32,
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="Outlet")

    # noslip
    noslip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                             batch_size_per_area=32,
                             criteria=(x>channel_length[0])&(x<channel_length[1]))
    self.add(noslip, name="NoSlip")

    # importance sampled interior
```

```
    # evaluate flow on grid for generating importance sampling probability function
    grid_res = [1024, 512]

    # make numpy function for computing the sdf
    sdf_fn = np_lambdify(geo.sdf,
                         ['x', 'y'])

    # make function for computing the network velocity derivatives
    # train domain now has access to sess, nets, and equations
    invar_placeholder = Variables.tf_variables(['x', 'y'],
                                               batch_size=grid_res[0]*grid_res[1])
    outvar_velocity_grad = unroll_graph(self.nets+self.equations,
                                        invar_placeholder,
                                        Key.convert_list(['u__x', 'u__y', 'v__x', 'v__y']))
    velocity_grad_fn = Variables.lambdify_np(invar_placeholder,
                                             outvar_velocity_grad, self.sess)

    # construct importance sampled constrain for interior
    class ImportanceSampledInterior(BC):
      def __init__(self):
        self.batch_size = 4000
        self.importance_prob_grid = None
        self.update_importance_prob_freq = 100
        self.sampled_counter = 0
        self.importance_prob_warmup = 25000
        self.epsilon_uniform = 0.

      def generate_importance_prob_grid(self):
        # sample 10 times as many points in batch to do adaptive sampling
        mesh_x, mesh_y = np.meshgrid(np.linspace(channel_length[0], channel_length[1], grid_res[0]),
                                     np.linspace(-outer_cylinder_radius, outer_cylinder_radius, grid_res[1]),
                                     indexing='ij')
        mesh_x = np.reshape(mesh_x, (grid_res[0]*grid_res[1], 1))
        mesh_y = np.reshape(mesh_y, (grid_res[0]*grid_res[1], 1))

        # compute sdf
        sdf = sdf_fn(x=mesh_x, y=mesh_y)

        # compute velocity grad
        velocity_grad = velocity_grad_fn(Variables({'x': mesh_x, 'y': mesh_y}))

        # compute probability density (zero if sdf is negative/outside domain)
        # probabiliy equal to the norm of the gradient
        dx = (channel_length[1] - channel_length[0]) / (grid_res[0]-1)
        dy = (2*outer_cylinder_radius) / (grid_res[1]-1)
        element_size = dx*dy
        vel_grad_norm = (velocity_grad['u__x']**2
                       + velocity_grad['u__y']**2
                       + velocity_grad['v__x']**2
                       + velocity_grad['v__y']**2)**0.5
        if self.sampled_counter <  self.importance_prob_warmup:
          unnormalized_prob = np.heaviside(sdf+max(dx,dy), 0.0) * (np.ones_like(sdf)) # uniform distribution
        else:
          unnormalized_prob = np.heaviside(sdf+max(dx,dy), 0.0) * (vel_grad_norm + self.epsilon_uniform)
        unnormalized_prob = np.reshape(unnormalized_prob, grid_res)

        # normalize probability distribution, (||prob|| = 1)
        area = (channel_length[1] - channel_length[0]) * (2*outer_cylinder_radius)
        normalized_prob = unnormalized_prob / np.sum(area * unnormalized_prob/ (grid_res[0]*grid_res[1]))

        # plot function for viewing
        plt.imshow(normalized_prob)
        plt.title("Sample Probability")
        plt.colorbar()
        plt.savefig("sample_prob.png")
        plt.close()

        return normalized_prob

      def invar_names(self):
        return ['x', 'y', 'area']

      def outvar_names(self):
        return ['continuity', 'momentum_x', 'momentum_y']

      def lambda_names(self):
        return ['lambda_continuity', 'lambda_momentum_x', 'lambda_momentum_y']

      def invar_fn(self, batch_size):
        # generate updated probability
```

```
        if self.sampled_counter == 0 or (self.sampled_counter % self.update_importance_prob_freq == 0 and self.
    sampled_counter > self.importance_prob_warmup):
          self.importance_prob_grid = self.generate_importance_prob_grid()
        self.sampled_counter += 1

        # sample points
        invar = {'x': np.zeros((0,1)),
                 'y': np.zeros((0,1)),
                 'area': np.zeros((0,1))}
        while True:
          # sample size (try to sample more then the batch so only do 1 loop)
          sample_size = 4*batch_size

          # sample points in range
          x = np.random.uniform(channel_length[0], channel_length[1], [sample_size, 1])
          y = np.random.uniform(-outer_cylinder_radius, outer_cylinder_radius, [sample_size, 1])
          rand = np.random.uniform(0, np.max(self.importance_prob_grid), [sample_size, 1])

          # compute index for interpolation
          dx = (channel_length[1] - channel_length[0]) / (grid_res[0]-1)
          dy = (2*outer_cylinder_radius) / (grid_res[1]-1)
          x_index = np.round((x - channel_length[0]) / dx).astype(dtype=np.int)
          y_index = np.round((y + outer_cylinder_radius) / dy).astype(dtype=np.int)

          # index importance grid
          prob = self.importance_prob_grid[x_index[:,0], y_index[:,0]]

          # remove points according to the rejection method (https://web.mit.edu/urban_or_book/www/book/
    chapter7/7.1.3.html)
          # also remove any point on edges with negative sdf
          sdf = sdf_fn(x=x, y=y)
          remove_criteria = np.logical_and(rand[:,0] < prob, sdf[:,0] > 0)
          x = x[remove_criteria, :]
          y = y[remove_criteria, :]
          prob = np.expand_dims(prob[remove_criteria], axis=-1)

          # add points to out dictionary
          invar['x'] = np.concatenate([invar['x'], x], axis=0)
          invar['y'] = np.concatenate([invar['y'], y], axis=0)
          invar['area'] = np.concatenate([invar['area'], prob], axis=0)

          # check if sampled enough points
          if invar['x'].shape[0] >= batch_size:
            invar['x'] = invar['x'][:batch_size]
            invar['y'] = invar['y'][:batch_size]
            invar['area'] = 1.0/(invar['area'][:batch_size]*batch_size)
            break

        return invar

      def outvar_fn(self, invar):
        outvar = {'continuity': np.zeros_like(invar['x']),
                  'momentum_x': np.zeros_like(invar['x']),
                  'momentum_y': np.zeros_like(invar['x'])}
        return outvar

      def lambda_fn(self, invar, outvar):
        lambda_weighting = {'lambda_continuity': sdf_fn(x=invar['x'], y=invar['y']) * invar['area'],
                            'lambda_momentum_x': sdf_fn(x=invar['x'], y=invar['y']) * invar['area'],
                            'lambda_momentum_y': sdf_fn(x=invar['x'], y=invar['y']) * invar['area']}
        return lambda_weighting

    # interior
    interior = ImportanceSampledInterior()
    self.add(interior, name="Interior")

    # make integral continuity
    outlet = geo.boundary_bc(outvar_sympy={'integral_continuity': 2},
                             batch_size_per_area=128,
                             lambda_sympy={'lambda_integral_continuity': 0.1},
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="OutletContinuity")
```

Listing 102: Defining a training domain with importance sampling.

12.8 Quasirandom Sampling

By default, Modulus generates the training points using a uniform distribution. Alternatively, one can use the Halton low-discrepancy sequence generator in Modulus (as described in Section 1.5.7). The following shows how to generate training points using a Halton sequence generator for the lid-driven cavity example by setting `quasirandom=True`.

```
class LDCTrain(TrainDomain):
  def __init__(self, **config):
    super(LDCTrain, self).__init__()

    #top wall
    topWall = geo.boundary_bc(outvar_sympy={'u': vel, 'v': 0},
                              batch_size_per_area=10000,
                              lambda_sympy={'lambda_u': 1.0 - 20 * Abs(x),  # weight edges to be zero
                                            'lambda_v': 1.0},
                              criteria=Eq(y, height / 2),
                              quasirandom=True)
    self.add(topWall, name="TopWall")

    # no slip
    bottomWall = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                                 batch_size_per_area=10000,
                                 criteria=y < height / 2,
                                 quasirandom=True)
    self.add(bottomWall, name="NoSlip")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0},
                               bounds={x: (-width / 2, width / 2),
                                       y: (-height / 2, height / 2)},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                                             'lambda_momentum_x': geo.sdf,
                                             'lambda_momentum_y': geo.sdf},
                               batch_size_per_area=400000,
                               quasirandom=True)
    self.add(interior, name="Interior")
```

Listing 103: Using Halton sequences to generate quasirandom training points for the lid-driven cavity example.

Please note that the Halton sequence generator in it's current form in Modulus should not be used for a geometry when specifying `fixed_var=False` for that geometry.

12.9 Example: Radial Basis Neural Networks

In this example we see the code for implementation of the RBF network and compare its results against the default Fully Connected network in Modulus.

The code for the RBF network can be found in the source code in `modulus/architecture/radial_basis.py`. Any new neural network architecture can be defined using the classes and modules used in the architectures defined in this directory. You can use any one of the pre-coded architecture as a template and make the required changes to define a new neural network. The code for the RBF neural network is shown below.

```
import tensorflow as tf
from ..config import str2bool
from ..arch import Arch
from ..variables import Variables
from ..tf_utils.activation_functions import set_nonlinearity
from ..tf_utils.layers import fc_layer, _variable

class RadialBasisArch(Arch):
  def __init__(self, **config):
    super(RadialBasisArch, self).__init__(**config)
    needed_config = RadialBasisArch.process_config(config)
    self.__dict__.update(needed_config)
    self.print_configs()
    self.bounds = None

  def set_bounds(self, bounds):
    self.bounds = bounds

  @classmethod
  def add_options(cls, group):
    group.add_argument('--nr_centers', help='number of radial basis', type=int,
                       default=128)
    group.add_argument('--sigma', help='sigma in kernel', type=float,
```

```
                          default=0.1)

  def _network_template(self, invars, out_names):
    # get input variables into tensor of shape [None, n]
    inputs = Variables.to_tensor(invars)

    # make center weight matrix
    assert self.bounds is not None, "Need to set bounds for raidal basis network"
    centers = []
    for dim in self.bounds:
      initializer = tf.initializers.random_uniform(minval=self.bounds[dim][0], maxval=self.bounds[dim][1])
      c = _variable('c_'+dim,
                    shape=[1,self.nr_centers,1],
                    initializer=initializer)
      centers.append(c)
    centers = tf.concat(centers, axis=-1)
    centers = tf.stop_gradient(centers)

    # tile centers and inputs so each tile runs on each input
    centers = tf.tile(centers,[tf.shape(inputs)[0],1,1])
    inputs = tf.expand_dims(inputs, axis=1)
    inputs = tf.tile(inputs,[1,tf.shape(centers)[1],1])

    # calc kernels
    #c = (1.0/(self.sigma*np.sqrt(2*np.pi)))
    radial_activation = tf.math.exp(-0.5*(tf.norm(centers-inputs, axis=-1)/self.sigma)**2)

    # perceptron out
    x = fc_layer(radial_activation, len(out_names), activation_fn=None, name='fc_final') # no weight norm on
     last layer
    return Variables.from_tensor(x, out_names)
```

Listing 104: Radial Basis Neural Network

We solve the Hemholtz equation in 2D defined in equation 109 [10].

$$\begin{aligned} \Delta u(x,y) + u(x,y) = q(x,y), (x,y) \in \Omega := (-1,1) \\ u(x,y) = 0, (x,y) \in \partial\Omega \end{aligned} \tag{109}$$

where,

$$q = -(\pi)^2 \sin(\pi x)\sin(4\pi y) - (4\pi)^2 \sin(\pi x)\sin(4\pi y) + \sin(\pi x)\sin(4\pi y) \tag{110}$$

The analytical solution to the above problem is given by equation 111

$$u = \sin(\pi x)\sin(4\pi y) \tag{111}$$

Note: The python script for the problem above, can be found at `examples/helmholtz/helmholtz_radial_basis.py`

We run the problem using the default fully connected architecture and the Radial Basis architecture shown above. Figure 60 shows the comparison of results between the two architectures.

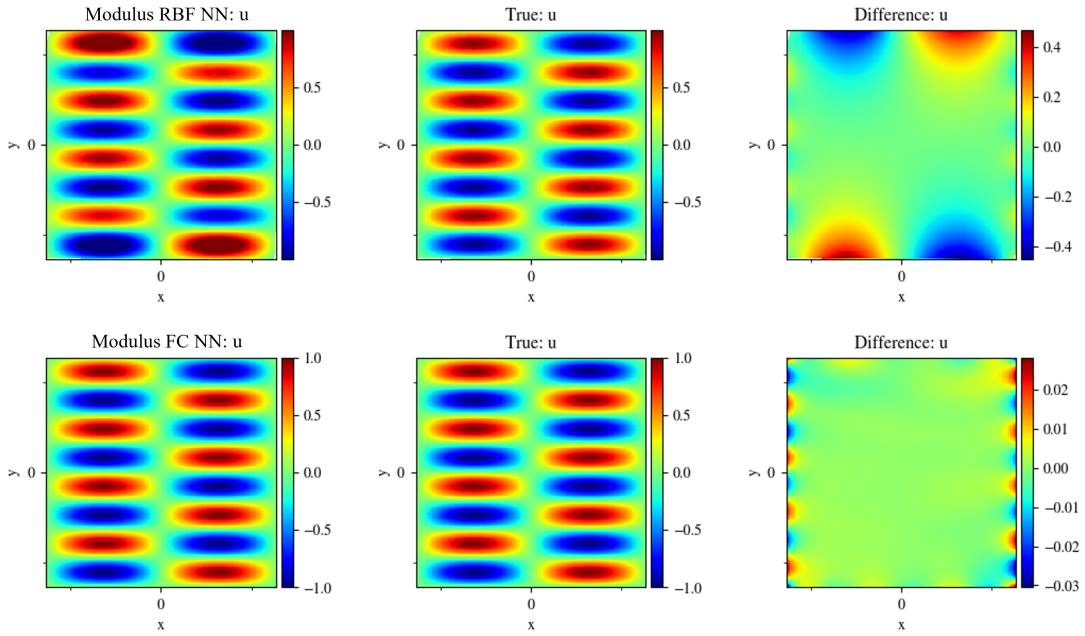


Figure 60: Radial Basis Network versus Fully connected Network

13 Multi-Physics Simulations: Conjugate Heat Transfer

13.1 Introduction

In this tutorial, we will use Modulus to study the conjugate heat transfer between the heat sink and the surrounding fluid. The temperature variations inside solid and fluid would be solved in a coupled manner with appropriate interface boundary conditions. In this tutorial, you would learn the following:

1. How to generate a 3D geometry using the geometry module in Modulus.
2. How to set up multiple point densities in different regions of the flow field.
3. How to set up a conjugate heat transfer problem using the interface boundary conditions in Modulus.
4. How to use the Muti-Phase training approach in Modulus for one-way coupled problems.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you to refer to tutorial 3 for information on creating monitor and inference domains and tutorial 8 for additional details on writing some of the thermal boundary conditions.

13.2 Problem Description

The geometry for a 3-fin heat sink placed inside a channel is shown in figure 61. The inlet to the channel is at 1 m/s. The pressure at the outlet is specified as 0 Pa. All the other surfaces of the geometry are treated as no-slip walls. The dimensions of the channel and the 3-fin heat sink are shown in figure 61.

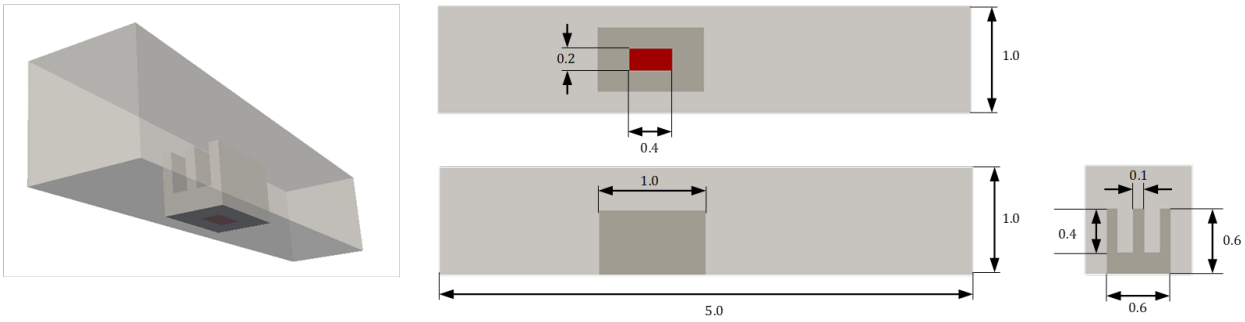


Figure 61: Three fin heat sink geometry (All dimensions in m)

The inlet is at 273.15 K. The channel walls are adiabatic. The heat sink has a heat source of 0.2x0.4 m at the bottom of the heat sink situated centrally on the bottom surface. The heat source generates heat such that the temperature gradient on the source surface is 360 K/m in the normal direction. Conjugate heat transfer takes place between the fluid-solid contact surface.

The properties fluid and thermal properties of the fluid and the solid are as follows:

Table 3: Fluid and Solid Properties

Property	Fluid	Solid
Kinematic Viscosity (m^2/s)	0.02	NA
Thermal Diffusivity (m^2/s)	0.02	0.0625
Thermal Conductivity $(W/m.K)$	1.0	5.0

13.3 Case Setup

In this tutorial, since we are dealing with only incompressible flow, there is a one-way coupling between the heat and flow equations. This means that it is possible to train the temperature field after the flow field is trained and

converged. Such an approach is useful while training the multi-physics problems which are one-way coupled as it is possible to achieve significant speed-up, as well as simulate cases with same flow boundary conditions but different thermal boundary conditions. One can easily use the same flow field as in input to train for different thermal boundary conditions.

Hence, for this problem we will have three separate files for domain, flow solver, and heat solver. The `three_fin_domain.py` will contain all the definitions of geometry and domains for flow as well as heat and the corresponding boundary conditions. `three_fin_flow_solver.py` would then take in information from the domain file and compute the flow field. Once the flow field is sufficiently converged, we can use `three_fin_heat_solver.py` to then take in the flow field generated and solve for the temperature distributions in fluid and solid simultaneously.

In this problem we will non-dimensionalize the temperature according to the following equation:

$$\theta = T/273.15 - 1.0 \tag{112}$$

Note: The python scripts for this problem can be found at `examples/three_fin_3d/laminar/`.

Importing the required packages

The list of the packages to be imported in each of the file are shown below. Only the relevant flow and heat domains/ classes are imported in `three_fin_flow_solver.py` and `three_fin_heat_solver.py` respectively.

```
from sympy import Symbol, Eq, tanh
import numpy as np
import tensorflow as tf

from modulus.dataset import TrainDomain, ValidationDomain, MonitorDomain
from modulus.data import Validation, Monitor
from modulus.sympy_utils.geometry_3d import Box, Channel, Plane
from modulus.csv_utils.csv_rw import csv_to_dict
```

Listing 105: Importing the required packages and modules for the domain file

```
# import domain
from three_fin_domain import ThreeFinFlowTrain, ThreeFinFlowMonitor, ThreeFinFlowVal, nu, rho

# import Modulus library
from modulus.solver import Solver
from modulus.PDES.navier_stokes import IntegralContinuity, NavierStokes
from modulus.controller import ModulusController
```

Listing 106: Importing the required packages and modules for the flow solver file

```
# import domain
from three_fin_domain import ThreeFinHeatTrain, ThreeFinHeatMonitor, ThreeFinHeatVal, D_solid, D_fluid, rho

# import Modulus library
from modulus.solver import Solver
from modulus.PDES.navier_stokes import GradNormal
from modulus.PDES.advection_diffusion import AdvectionDiffusion, IntegralAdvection
from modulus.PDES.diffusion import Diffusion, DiffusionInterface
from modulus.controller import ModulusController
```

Listing 107: Importing the required packages and modules for the heat solver file

13.3.1 Creating Geometry

As described earlier, we will edit the `three_fin_domain.py` file to create the geometry. We will use the `Box` primitive to create the 3-fin geometry and `Channel` primitive to generate the channel. Similar to 2D, `Channel` and `Box` are defined by using the two corner points. Like 2D, the `Channel` geometry has no bounding planes in the x-direction. We will also make use of the `repeat` attribute to create the fins. This attribute speeds up the generation of repetitive structures in comparison to constructing the same geometry using for/while loop.

We will use the `Plane` geometry to create the planes at the inlet and outlet. The code for generating the required geometries is shown below. Please note the normal directions for the inlet and outlet planes.

Additionally, the parameters required for solving the heat part as also defined upfront, ex. dimensions and locations of source etc.

```
# geometry params for domain
channel_origin        = (-2.5, -0.5, -0.5)
channel_dim           = (5.0, 1.0, 1.0)
heat_sink_base_origin = (-1.0, -0.5, -0.3)
heat_sink_base_dim    = (1.0, 0.2, 0.6)
fin_origin            = heat_sink_base_origin
fin_dim               = (1.0, 0.6, 0.1)
total_fins            = 3
flow_box_origin       = (-1.1, -0.5, -0.5)
flow_box_dim          = (1.2, 1.0, 1.0)
source_origin         = (-0.7, -0.5, -0.1)
source_dim            = (0.4, 0.0, 0.2)
source_area           = 0.08

# params for simulation
# fluid params
nu                    = 0.02
rho                   = 1
inlet_vel             = 1.0
volumetric_flow       = 1.0
# heat params
D_solid               = 0.0625
D_fluid               = 0.02
inlet_t               = 293.15
grad_t                = 360
inlet_t = inlet_t/273.15 - 1.0
grad_t = grad_t/273.15

# define sympy varaibles to parametize domain curves
x, y, z = Symbol('x'), Symbol('y'), Symbol('z')

# define geometry
# channel
channel = Channel(channel_origin, (channel_origin[0]+channel_dim[0],
                                   channel_origin[1]+channel_dim[1],
                                   channel_origin[2]+channel_dim[2]))

# three fin heat sink
heat_sink_base = Box(heat_sink_base_origin, (heat_sink_base_origin[0]+heat_sink_base_dim[0], # base of heat
    sink
                                             heat_sink_base_origin[1]+heat_sink_base_dim[1],
                                             heat_sink_base_origin[2]+heat_sink_base_dim[2]))
fin_center = (fin_origin[0] + fin_dim[0]/2, fin_origin[1] + fin_dim[1]/2, fin_origin[2] + fin_dim[2]/2)
fin = Box(fin_origin, (fin_origin[0]+fin_dim[0],
                       fin_origin[1]+fin_dim[1],
                       fin_origin[2]+fin_dim[2]))
gap = (heat_sink_base_dim[2]-fin_dim[2])/(total_fins-1) # gap between fins
fin.repeat(gap, repeat_lower=(0,0,0), repeat_higher=(0,0,total_fins-1), center=fin_center)
three_fin = heat_sink_base + fin

# entire geometry
geo = channel - three_fin


# inlet and outlet
inlet = Plane(channel_origin, (channel_origin[0], channel_origin[1]+channel_dim[1], channel_origin[2]+
    channel_dim[2]), -1)
outlet = Plane((channel_origin[0]+channel_dim[0], channel_origin[1], channel_origin[2]), (channel_origin[0]+
    channel_dim[0], channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]), 1)
```

Listing 108: Generating the geometry

13.3.1.1 Varying point sampling density & adding parameterized integral continuity planes

In this problem, we will sample points at a higher density in the vicinity of the heat sink. This will help in achieving a faster convergence and also higher accuracy as majority of the flow field variations are expected to occur in the vicinity of the heat sink. To achieve this, we will create an high resolution `flow_box` surrounding the heat sink which will be used to sample the points more densely. This high resolution box would be then subtracted out of the channel geometry to create the low resolution box.

We will also create some integral continuity planes in the vicinity of the heat sink to help the development of the flow field faster. Unlike tutorial 8 where we placed the planes at static location, we will parameterize them such that a new plane is drawn every training iteration. These planes will enforce the constraint of satisfying a specified mass flow through these planes. These parameterized planes would be used to set the `IntegralContinuity` condition in the equations and solver parts of the code. To generate these parameterized planes, a symbolic variable for x position of the

plane would be created and we will randomly sample planes in the range (-1.1, 0.1) which is $\pm$ 0.1 m in the aft and rear of the heat sink in x direction. In this section we will only define the planes in a symbolic fashion and specify the ranges for the `x_pos`. Actual sampling of the planes would be done during the boundary condition definition.

The code to generate a separate `flow_box` and parameterized `integral_plane` can be found below.

Note: The addition of integral continuity planes and separate flow box for dense sampling are optional. However, they have been found to improve the accuracy and convergence to a great extent and it is recommended to enforce such techniques for any complex flow phenomena.

```
# low and high resultion geo away and near the heat sink
flow_box = Box(flow_box_origin, (flow_box_origin[0]+flow_box_dim[0], # base of heat sink
                                 flow_box_origin[1]+flow_box_dim[1],
                                 flow_box_origin[2]+flow_box_dim[2]))
lr_geo = geo - flow_box
hr_geo = geo & flow_box
lr_bounds_x = (channel_origin[0], channel_origin[0] + channel_dim[0])
lr_bounds_y = (channel_origin[1], channel_origin[1] + channel_dim[1])
lr_bounds_z = (channel_origin[2], channel_origin[2] + channel_dim[2])
hr_bounds_x = (flow_box_origin[0], flow_box_origin[0] + flow_box_dim[0])
hr_bounds_y = (flow_box_origin[1], flow_box_origin[1] + flow_box_dim[1])
hr_bounds_z = (flow_box_origin[2], flow_box_origin[2] + flow_box_dim[2])

# planes for integral continuity
x_pos = Symbol('x_pos')
integral_plane = Plane((x_pos, channel_origin[1], channel_origin[2]),
                       (x_pos, channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]),
                       1)
x_pos_range = {x_pos: lambda batch_size: np.full((batch_size, 1), np.random.uniform(-1.1, 0.1))}
```

Listing 109: Adding geometries for dense point sampling and integral continuity planes

13.3.2 Defining the Flow Boundary conditions and Equations

For the multi-phase training approach, we will generate the `ThreeFinFlowTrain` containing all the flow boundary conditions and `ThreeFinHeatTrain` containing all the thermal boundary conditions separately.

The contents of `ThreeFinFlowTrain` are listed below.

Inlet, Outlet and Channel and Heat Sink walls For inlet boundary conditions, we will specify the velocity to be a constant velocity of 1.0 m/s in x-direction. Like in tutorial 2, we will weight the velocity by the SDF of inlet plane to avoid sharp discontinuity at the boundaries. For outlet, we will specify the pressure to be 0. All the channel walls and heat sink walls are treated as no slip boundaries.

Interior The flow equations can be specified in the low resolution and high resolution domains of the problem by using `interior_bc`. We will use different `batch_size_per_area` for each of these domains (`interiorF_lr` and `interiorF_hr`) to vary the density of points.

Integral Continuity The integral continuity planes would be sampled by using a for loop as shown in the code. The inlet volumetric flow is 1 m^3/s and hence we will specify `integral_continuity` variable as 1.0 on each of these planes.

```
# define flow domain
class ThreeFinFlowTrain(TrainDomain):
  def __init__(self, **config):
    super(ThreeFinFlowTrain, self).__init__()
    # inlet
    inletBC = inlet.boundary_bc(outvar_sympy={'u': inlet_vel, 'v': 0, 'w': 0},
                                batch_size_per_area=500,
                                lambda_sympy={'lambda_u': channel.sdf, # weight zero on edges
                                              'lambda_v': 1.0,
                                              'lambda_w': 1.0},
                                criteria=Eq(x, channel_origin[0]))
    self.add(inletBC, name="Inlet")

    # outlet
    outletBC = outlet.boundary_bc(outvar_sympy={'p': 0},
                                  batch_size_per_area=500,
                                  criteria=Eq(x, channel_origin[0]+channel_dim[0]))
    self.add(outletBC, name="Outlet")
```

```
    # no slip for channel walls
    no_slip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                              batch_size_per_area=500)
    self.add(no_slip, name="NoSlipChannel")

    # flow interior low res away from three fin
    interiorF_lr = lr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                      bounds={x: lr_bounds_x, y: lr_bounds_y, z: lr_bounds_z},
                                      lambda_sympy={'lambda_continuity': geo.sdf,
                                                    'lambda_momentum_x': geo.sdf,
                                                    'lambda_momentum_y': geo.sdf,
                                                    'lambda_momentum_z': geo.sdf},
                                      batch_size_per_area=500)
    self.add(interiorF_lr, name="FlowInterior_LR")

    # flow interiror high res near three fin
    interiorF_hr = hr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                      bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                      lambda_sympy={'lambda_continuity': geo.sdf,
                                                    'lambda_momentum_x': geo.sdf,
                                                    'lambda_momentum_y': geo.sdf,
                                                    'lambda_momentum_z': geo.sdf},
                                      batch_size_per_area=3000)
    self.add(interiorF_hr, name="FlowInterior_HR")

    # integral continuity
    for i in range(5):
      IC = integral_plane.boundary_bc(outvar_sympy={'integral_continuity': 1.0},
                                      batch_size_per_area=10000,
                                      lambda_sympy={'lambda_integral_continuity': 1.0},
                                      criteria=geo.sdf>0,
                                      param_ranges=x_pos_range,
                                      fixed_var=False)
      self.add(IC, name="IntegralContinuity_"+str(i))
```

Listing 110: Defining the flow boundary conditions and equations

13.3.3 Defining the Thermal Multi-Phase Boundary conditions and Equations

The contents of `ThreeFinHeatTrain` are described below.

Inlet, Outlet and Channel walls: For the heat part, we will specify temperature at the inlet. All the outlet and the channel walls will have a zero gradient boundary condition which will be enforced by setting `'normal_gradient_theta_f'` equal to 0. We will use `'theta_f'` for defining the temperatures in fluid and `'theta_s'` for defining the temperatures in solid.

Fluid and Solid Interior: Just like tutorial 8, we will set `'advection_diffusion'` equal to 0 in both, low and high resolution fluid domains. For solid interior, we will set `'diffusion'` equal to 0.

Fluid-Solid Interface: For the interface between fluid and solid, we will enforce both Neumann and Dirchlet boundary condition by setting `'diffusion_interface_dirichlet_theta_f_theta_s'` and `'diffusion_interface_neumann_theta_f_theta_s'` both equal to 0.

Note: The order in which you define `'theta_f'` and `'theta_s'` in the interface boundary condition is important. The different conductivities for fluid and solid will be specified in the same order during the heat solver file definition.

Heat Source: We will apply a $tanh$ smoothing while defining the heat source on the bottom wall of the heat sink. We have found that smoothing out the sharp boundaries helps in training the Neural Network converge faster. The `'normal_gradient_theta_s'` is set equal to `grad_t` in the area of source and 0 everywhere else on the bottom surface of heat sink.

Integral Advection: In addition to the regular boundary conditions, like integral continuity, we will impose the conservation of heat flux in an integral form on the outlet plane using `'integral_advection'` keyword. The heat flux out of the system must be equal to the influx which is the total of flux entering through the inlet and the source. For

this problem, applying such balance give us the following:

$$\int \rho_{fluid} c_{fluid} T_{out} U_{out} ds = \rho_{fluid} c_{fluid} T_{in} A_{in} U_{in} + k_{solid} (dT/dn) A_{source} \tag{113}$$

$$\int T_{out} U_{out} ds = T_{in} A_{in} U_{in} + \frac{k_{solid} (dT/dn) A_{source}}{\rho_{fluid} c_{fluid}} \tag{114}$$

```
class ThreeFinHeatTrain(TrainDomain):
  def __init__(self, **config):
    super(ThreeFinHeatTrain, self).__init__()
    # inlet
    inletBC = inlet.boundary_bc(outvar_sympy={'theta_f': inlet_t},
                                batch_size_per_area=500,
                                criteria=Eq(x, channel_origin[0]))
    self.add(inletBC, name="Inlet")

    # outlet
    outletBC = outlet.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                  batch_size_per_area=500,
                                  criteria=Eq(x, channel_origin[0]+channel_dim[0]))
    self.add(outletBC, name="Outlet")

    # channel walls insulating
    walls = channel.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                batch_size_per_area=500,
                                criteria=three_fin.sdf<-1e-5)
    self.add(walls, name="ChannelWalls")

    # flow interior low res away from three fin
    interiorF_lr = lr_geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                      bounds={x: lr_bounds_x, y: lr_bounds_y, z: lr_bounds_z},
                                      batch_size_per_area=500)
    self.add(interiorF_lr, name="FlowInterior_LR")

    # flow interiror high res near three fin
    interiorF_hr = hr_geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                      bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                      batch_size_per_area=3000)
    self.add(interiorF_hr, name="FlowInterior_HR")

    # solid interior
    interiorS = three_fin.interior_bc(outvar_sympy={'diffusion_theta_s': 0},
                                  bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                  batch_size_per_area=30000,
                                  lambda_sympy={'lambda_diffusion_theta_s': 100})
    self.add(interiorS, name="SolidInterior")

    # fluid solid interface
    interface = three_fin.boundary_bc(outvar_sympy={'diffusion_interface_dirichlet_theta_f_theta_s': 0, '
     diffusion_interface_neumann_theta_f_theta_s': 0},
                                  batch_size_per_area=500,
                                  criteria=channel.sdf>0)
    self.add(interface, name="Interface")

    # heat source
    sharpen_tanh = 60.0
    source_func_xl = (tanh(sharpen_tanh*(x - source_origin[0])) + 1.0)/2.0
    source_func_xh = (tanh(sharpen_tanh*((source_origin[0]+source_dim[0]) - x)) + 1.0)/2.0
    source_func_zl = (tanh(sharpen_tanh*(z - source_origin[2])) + 1.0)/2.0
    source_func_zh = (tanh(sharpen_tanh*((source_origin[2]+source_dim[2]) - z)) + 1.0)/2.0
    gradient_normal = grad_t * source_func_xl * source_func_xh * source_func_zl * source_func_zh
    heat_source = three_fin.boundary_bc(outvar_sympy={'normal_gradient_theta_s': gradient_normal},
                                    batch_size_per_area=5000,
                                    criteria=Eq(y, source_origin[1]))
    self.add(heat_source, name="HeatSource")

    # integral advection diffusion
    IC = outlet.boundary_bc(outvar_sympy={'integral_advection_theta_f': 5.0*source_area*grad_t/50 + inlet_t*1.0*
     inlet_vel},
                            lambda_sympy={'lambda_integral_advection_theta_f': 0.1},
                            batch_size_per_area=10000)
    self.add(IC, name="IntegralAdvection")
```

Listing 111: Defining the thermal boundary conditions and equations

13.3.4 Creating Validation and Monitor domains

Like separate training domains, we will create separate validation, and monitor domains for flow and heat part. In this tutorial we will monitor the residuals of flow equations and pressure drops during the flow field simulation and monitor the residuals of heat equations and peak temperature reached in the source chip during the heat simulation in the second phase.

```
# validation data
# flow data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z',
           'U:0': 'u', 'U:1': 'v','U:2': 'w', 'p_rgh': 'p', 'T': 'theta_f'}
openfoam_var = csv_to_dict('../openfoam/threeFin_extend_fluid0.csv', mapping)
openfoam_var['theta_f'] = openfoam_var['theta_f']/273.15 - 1.0 # normalize heat
openfoam_var['x'] = openfoam_var['x'] + channel_origin[0]
openfoam_var['y'] = openfoam_var['y'] + channel_origin[1]
openfoam_var['z'] = openfoam_var['z'] + channel_origin[2]
openfoam_invar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['x','y','z']}
openfoam_outvar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['u','v','w','p']}
openfoam_outvar_flow_heat_numpy = {key: value for key, value in openfoam_var.items() if key in ['theta_f']}
# solid data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z',
           'T': 'theta_s'}
openfoam_var = csv_to_dict('../openfoam/threeFin_extend_solid0.csv', mapping)
openfoam_var['theta_s'] = openfoam_var['theta_s']/273.15 - 1.0 # normalize heat
openfoam_var['x'] = openfoam_var['x'] + channel_origin[0]
openfoam_var['y'] = openfoam_var['y'] + channel_origin[1]
openfoam_var['z'] = openfoam_var['z'] + channel_origin[2]
openfoam_invar_solid_numpy = {key: value for key, value in openfoam_var.items() if key in ['x','y','z']}
openfoam_outvar_solid_numpy = {key: value for key, value in openfoam_var.items() if key in ['theta_s']}

class ThreeFinFlowVal(ValidationDomain):
  def __init__(self, **config):
    super(ThreeFinFlowVal, self).__init__()
    # fluid validation
    val = Validation.from_numpy(openfoam_invar_flow_numpy, openfoam_outvar_flow_numpy)
    self.add(val, name='ValFlowField')

class ThreeFinHeatVal(ValidationDomain):
  def __init__(self, **config):
    super(ThreeFinHeatVal, self).__init__()
    # heat in fluid validation
    val = Validation.from_numpy(openfoam_invar_flow_numpy, openfoam_outvar_flow_heat_numpy)
    self.add(val, name='ValFlowHeat')

    # heat in solid validation
    val = Validation.from_numpy(openfoam_invar_solid_numpy, openfoam_outvar_solid_numpy)
    self.add(val, name='ValSolidHeat')
```

Listing 112: Defining the Validation domain

```
class ThreeFinFlowMonitor(MonitorDomain):
  def __init__(self, **config):
    super(ThreeFinFlowMonitor, self).__init__()
    # metric for equation residuals
    global_monitor = Monitor(geo.sample_interior(10000, bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z})
     ,
                             {'mass_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['continuity'])),

                              'momentum_x_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_x'])),
                              'momentum_y_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_y'])),
                              'momentum_z_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_z']))})
    self.add(global_monitor, 'ResiduallMonitor')

    # metric for pressure drop front
    p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges={x_pos: heat_sink_base_origin[0]-
     heat_sink_base_dim[0]}),
                        {'pressure': lambda var: tf.reduce_mean(var['p'])})
    self.add(p_monitor, 'FrontPressure')
    p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges={x_pos: heat_sink_base_origin[0]+2*
     heat_sink_base_dim[0]}),
                        {'pressure': lambda var: tf.reduce_mean(var['p'])})
    self.add(p_monitor, 'BackPressure')

class ThreeFinHeatMonitor(MonitorDomain):
  def __init__(self, **config):
```

```
    super(ThreeFinHeatMonitor, self).__init__()
    # metric for equation residuals
    global_monitor = Monitor(geo.sample_interior(10000, bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z})
    ,
                             {'advection_diffusion': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     advection_diffusion']))})
    self.add(global_monitor, 'FlowResiduallMonitor')
    global_monitor = Monitor(three_fin.sample_interior(100000, bounds={x: hr_bounds_x, y: hr_bounds_y, z:
     hr_bounds_z}),
                             {'diffusion': lambda var: tf.reduce_sum(var['area']*tf.abs(var['diffusion_theta_s'])
     )})
    self.add(global_monitor, 'SolidResiduallMonitor')

    # metric for peak temp
    temp_monitor = Monitor(three_fin.sample_boundary(10000, criteria=Eq(y, source_origin[1])),
                           {'peak_temp': lambda var: tf.reduce_max(var['theta_s'])})
    self.add(temp_monitor, 'PeakTempMonitor')
```

Listing 113: Defining the monitors

13.4 Making the neural network, Multi-Phase training

Once all the domain definitions are completed in the `three_fin_domain.py` file, we can complete the `three_fin_flow_solver.py` to add all the required equations and run the Modulus solver. This part of the code is similar to what we have seen in previous tutorials. `three_fin_flow_solver.py` can be found below.

```
class ThreeFinFlowSolver(Solver):
  train_domain = ThreeFinFlowTrain
  monitor_domain = ThreeFinFlowMonitor
  val_domain = ThreeFinFlowVal

  def __init__(self, **config):
    super(ThreeFinFlowSolver, self).__init__(**config)
    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_three_fin_fluid_flow',
        'rec_results_cpu': True,
        'max_steps': 500000,
        'decay_steps': 5000,
        })

if __name__ == '__main__':
  ctr = ModulusController(ThreeFinFlowSolver)
  ctr.run()
```

Listing 114: Defining flow solver

Once the flow part is trained, we will use that converged checkpoint to initialize the training for the heat part. This can be done by specifying the fluid flow's network directory as an initializer for the heat simulation. To achieve this, we specify the path for fluid flow network directory to the `'initialize_network_dir'` key in `update_defaults` function.

Also, for making the multi-phase approach to work, you will need to maintain the exact same flow network in the heat solver file as well. Thus, in the heat solver file, along with creating two networks for solving the heat in the solid and fluid, we will create the same flow network as used in the flow solver file.

Relevant equations are added in a similar way as seen in the flow solver file. Please note the order of `'theta_f'` and `'theta_s'` while specifying the `DiffusionInterface` equation. The order must be same as used in train domain's definition. The following two variables are the conductivites for fluid and solid respectively. Similar to tutorial 8, we will stop the gradients for velocity components in the `AdvectionDiffusion` equation.

The code for the `three_fin_heat_solver.py` can be found below.

```
class ThreeFinHeatSolver(Solver):
  train_domain = ThreeFinHeatTrain
```

```
  monitor_domain = ThreeFinHeatMonitor
  val_domain = ThreeFinHeatVal

  def __init__(self, **config):
    super(ThreeFinHeatSolver, self).__init__(**config)
    self.equations = (AdvectionDiffusion(T='theta_f', rho=rho, D=D_fluid, dim=3, time=False).make_node(
     stop_gradients=['u', 'v', 'w'])
                      + Diffusion(T='theta_s', D=D_solid, dim=3, time=False).make_node()
                      + DiffusionInterface('theta_f', 'theta_s', 1.0, 5.0, dim=3, time=False).make_node()
                      + GradNormal('theta_f', dim=3, time=False).make_node()
                      + GradNormal('theta_s', dim=3, time=False).make_node()
                      + IntegralAdvection(T='theta_f', dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    solid_net = self.arch.make_node(name='solid_net',
                                    inputs=['x', 'y', 'z'],
                                    outputs=['theta_s'])
    flow_heat_net = self.arch.make_node(name='flow_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['theta_f'])
    self.nets = [flow_net, solid_net, flow_heat_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_three_fin_heat',
        'initialize_network_dir': './network_checkpoint_three_fin_fluid_flow',
        'rec_results_cpu': True,
        'max_steps': 500000,
        'decay_steps': 5000,
        })

if __name__ == '__main__':
  ctr = ModulusController(ThreeFinHeatSolver)
  ctr.run()
```

Listing 115: Defining the heat solver

13.5 Running the Modulus Solver

Once both the solver files are defined, we will run the `three_fin_flow_solver.py` first to solve for the flow field. Once a desired level of convergence is achieved, we can end the simulation and run the `three_fin_heat_solver.py` file.

13.6 Results and Post-processing

Table 4 shows the results of Pressure drop and Peak temperatures obtained from the Modulus and compares it with the results from OpenFOAM solver.

Table 4: Comparisons of Results with OpenFOAM

	Modulus	OpenFOAM
Pressure Drop (Pa)	7.51	7.49
Peak Temperature $(^{\circ}C)$	78.35	78.05

13.6.1 Plotting gradient quantities: Wall Velocity Gradients

In a variety of applications, it is desirable to plot the gradients of some quantities inside the domain. One such example relevant to fluid flows is the wall velocity gradients and wall shear stresses. These quantities are often plotted to compute frictional forces, etc. We can visualize such quantities in Modulus by outputting the x, y and z derivatives of the desired variables using an `Inference` domain. Below we show an example of how to create such a inference domain for outputting velocity gradients on the walls of the domain.

```
class ThreeFinFlowInference(InferenceDomain):
  def __init__(self, **config):
    super(ThreeFinFlowInference, self).__init__()
    # Inference domain for wall velocity gradients
```

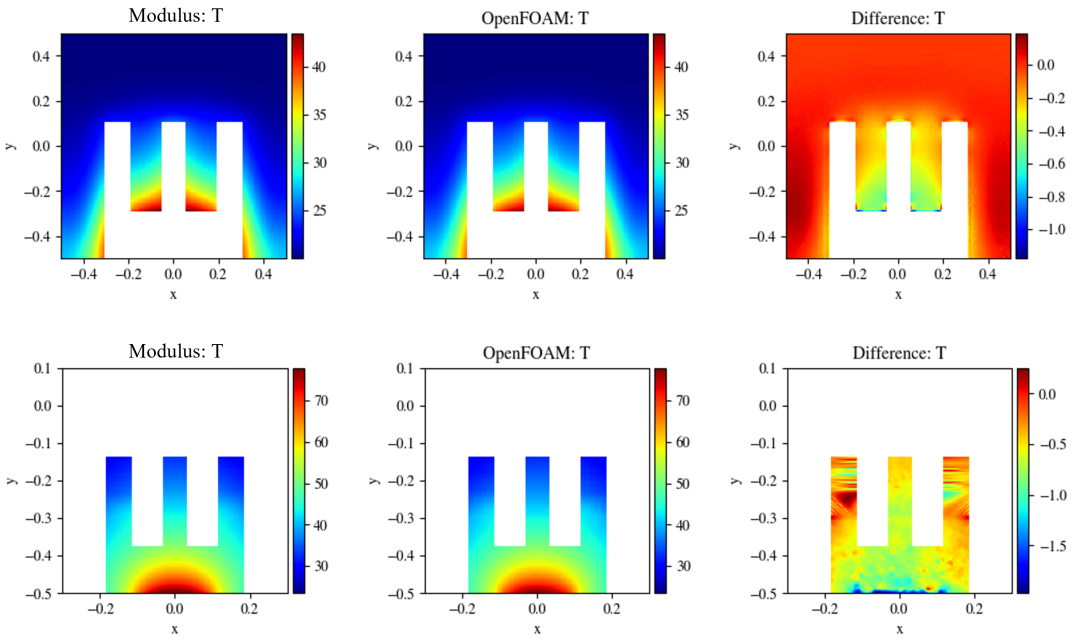


Figure 62: Left: Modulus. Center: OpenFOAM. Right: Difference. Top: Temperature distribution in Fluid. Bottom: Temperature distribution in Solid (*Temperature scales in $^{\circ}C$*)

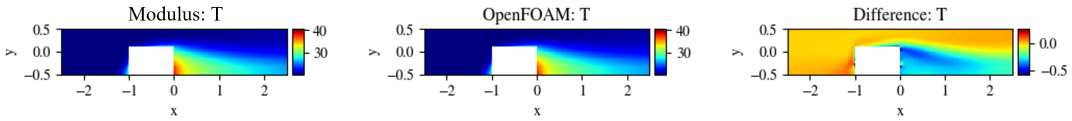


Figure 63: Left: Modulus. Center: OpenFOAM. Right: Difference. (*Temperature scales in $^{\circ}C$*)

```
    walls = Inference(geo.sample_boundary(100000), ['u__x', 'u__y', 'u__z',
                                                    'v__x', 'v__y', 'v__z',
                                                    'w__x', 'w__y', 'w__z'])

    self.add(walls, name='WallGradients')
```

Listing 116: Post-processing gradient quantities

As we have seen, such domains can also be added as a post-processing step, after training and then the script can be executed with `--run_mode=eval` flag to just run the inference.

Then, we can post-process these quantities based on our choice to visualize the desired variables using Paraview's Calculator Filter https://kitware.github.io/paraview-docs/latest/python/paraview.simple.Calculator.html. The wall velocity gradients comparison between OpenFOAM and Modulus is shown in Figure 64.

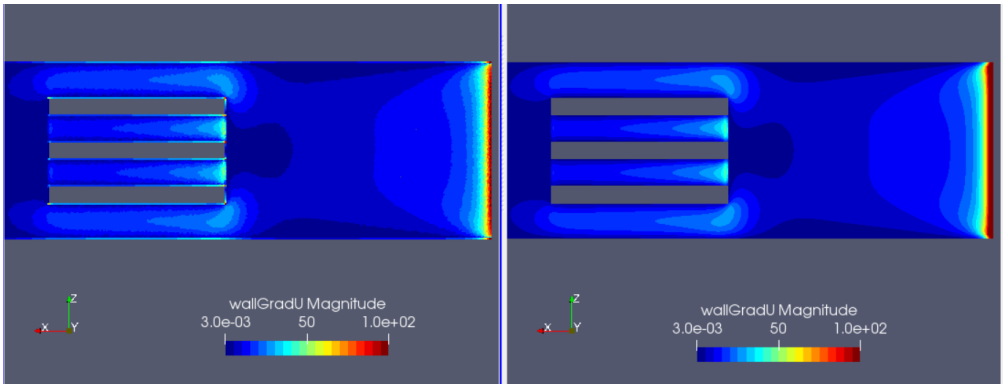


Figure 64: Comparison of magnitude of wall velocity gradients. Left: Modulus. Right: OpenFOAM

14 Forward simulation using STL geometry: Blood Flow in Intracranial Aneurysm

14.1 Introduction

In this tutorial, we will import an STL file for a complicated geometry and use Modulus' SDF library to sample points on the surface and the interior of the STL and train the PINNs to predict flow in this complex geometry. In this tutorial you will learn the following:

1. How to import an STL file in Modulus and sample points in the interior and on the surface of the geometry.

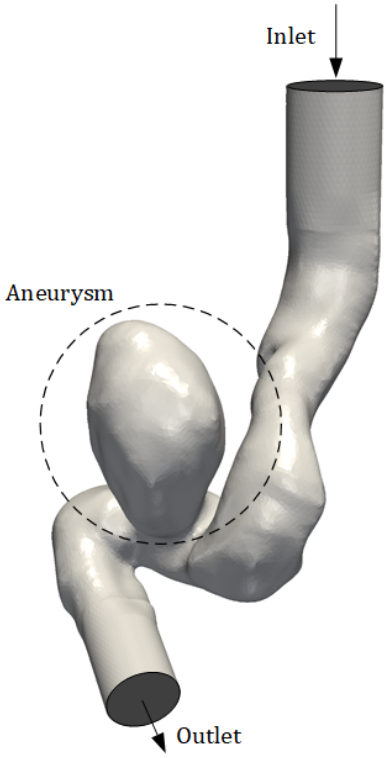


Figure 65: Aneurysm STL file

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of the Modulus user interface. Also, we recommend you to refer to tutorial 3 for information on creating monitor and inference domains. Additionally, to use the modules described in this tutorial, make sure your system satisfies the requirements for SDF library mentioned in Table 1.

Note: For the interior sampling to work, ensure that the STL geometry is watertight. This requirement is not necessary for sampling points on the surface.

14.2 Problem Description

For this simulation we apply a no-slip boundary condition on the walls of the aneurysm $u, v, w = 0$. For the inlet we use a parabolic flow where the flow goes in the normal direction of the inlet and has peak velocity 1.5. The outlet has a zero pressure condition, $p = 0$. The kinematic viscosity of the fluid is 0.025 and the density is a constant 1.0.

14.3 Case Setup

In this tutorial, we will use Modulus' `Mesh` module to sample points using a STL geometry. The module works similar to Modulus' geometry module with methods like `interior_bc`, `boundary_bc` to sample points in the interior and the boundary of the geometry. Currently, we would require separate STL files for each boundary of the geometry and another watertight geometry for sampling points in the interior of the geometry.

Note: All the python scripts for this problem can be found at `examples/aneurysm/`.

Importing the required packages

The list of required packages can be found below. We will import Modulus' `Mesh` module to sample points on the STL geometry.

```
import copy
from sympy import Symbol, sqrt, Max
import numpy as np
import tensorflow as tf

from modulus.solver import Solver
from modulus.dataset import TrainDomain, ValidationDomain, InferenceDomain, MonitorDomain
from modulus.data import BC, Validation, Inference, Monitor
from modulus.mesh_utils.mesh import Mesh
from modulus.PDES.navier_stokes import IntegralContinuity, NavierStokes
from modulus.controller import ModulusController
from modulus.csv_utils.csv_rw import csv_to_dict
```

Listing 117: Importing the required packages and modules

14.3.1 Using STL files to create Train domain

The STL geometries can be imported using the `Mesh.from_stl()` function. This function takes in the path of the STL geometry as input. We will need to specify the value of attribute `airtight` as `False` for the open surfaces (eg. boundary STL files).

Then these mesh objects can be sampled either on surface or interior similar to tutorial 2 using the `boundary_bc` or `interior_bc` functions. **Note:** For this tutorial, we will normalize the geometry by scaling it and centering it about the origin (0, 0, 0). This will help in speeding up the training process. Also, we will define a custom function for generating parabolic inlet for the blood vessel.

The code to sample using STL geometry, define all these functions, boundary conditions and generate the `TrainDomain` is shown below.

```
# read stl files to make meshes
point_path = './stl_files/'
inlet_mesh = Mesh.from_stl(point_path + 'aneurysm_inlet.stl', airtight=False)
outlet_mesh = Mesh.from_stl(point_path + 'aneurysm_outlet.stl', airtight=False)
noslip_mesh = Mesh.from_stl(point_path + 'aneurysm_noslip.stl', airtight=False)
integral_mesh = Mesh.from_stl(point_path + 'aneurysm_integral.stl', airtight=False)
interior_mesh = Mesh.from_stl(point_path + 'aneurysm_closed.stl')

# params
nu = 0.025
inlet_vel = 1.5

# inlet velocity profile
def circular_parabola(x, y, z, center, normal, radius, max_vel):
  centered_x = x-center[0]
  centered_y = y-center[1]
  centered_z = z-center[2]
  distance = sqrt(centered_x**2 + centered_y**2 + centered_z**2)
  parabola = max_vel*Max((1 - (distance/radius)**2), 0)
  return normal[0]*parabola, normal[1]*parabola, normal[2]*parabola

# normalize meshes
def normalize_mesh(mesh, center, scale):
  mesh.translate([-c for c in center])
  mesh.scale(scale)

# normalize invars
def normalize_invar(invar, center, scale, dims=2):
  invar['x'] -= center[0]
  invar['y'] -= center[1]
```

```
  invar['z'] -= center[2]
  invar['x'] *= scale
  invar['y'] *= scale
  invar['z'] *= scale
  if 'area' in invar.keys():
    invar['area'] *= scale**dims
  return invar

# scale and normalize mesh and openfoam data
center = (-18.40381048596882, -50.285383353981196, 12.848136936899031)
scale = 0.4
normalize_mesh(inlet_mesh, center, scale)
normalize_mesh(outlet_mesh, center, scale)
normalize_mesh(noslip_mesh, center, scale)
normalize_mesh(integral_mesh, center, scale)
normalize_mesh(interior_mesh, center, scale)
openfoam_invar = normalize_invar(openfoam_invar, center, scale, dims=3)

# geom params
inlet_normal = (0.8526, -0.428, 0.299)
inlet_area = 21.1284*(scale**2)
inlet_center = (-4.24298030045776, 4.082857101816247, -4.637790193399717)
inlet_radius = np.sqrt(inlet_area/np.pi)
outlet_normal = (0.33179, 0.43424, 0.83747)
outlet_area = 12.0773*(scale**2)
outlet_radius = np.sqrt(outlet_area/np.pi)

# define domain
class AneurysmTrain(TrainDomain):
  def __init__(self, **config):
    super(AneurysmTrain, self).__init__()
    # Inlet
    u, v, w = circular_parabola(Symbol('x'),
                                Symbol('y'),
                                Symbol('z'),
                                center=inlet_center,
                                normal=inlet_normal,
                                radius=inlet_radius,
                                max_vel=inlet_vel)
    inlet = inlet_mesh.boundary_bc(outvar_sympy={'u': u, 'v': v, 'w': w},
                                   batch_size_per_area=256)
    self.add(inlet, name="Inlet")

    # Outlet
    outlet = outlet_mesh.boundary_bc(outvar_sympy={'p': 0},
                                     batch_size_per_area=256)
    self.add(outlet, name="Outlet")

    # Noslip
    noslip = noslip_mesh.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                                     batch_size_per_area=32)
    self.add(noslip, name="Noslip")

    # Interior
    interior = interior_mesh.interior_bc(outvar_sympy={'continuity': 0,
                                                       'momentum_x': 0,
                                                       'momentum_y': 0,
                                                       'momentum_z': 0},
                                         batch_size_per_area=128,
                                         batch_per_epoch=1000)
    self.add(interior, name="Interior")

    # Integral Continuity 1
    ic_1 = outlet_mesh.boundary_bc(outvar_sympy={'integral_continuity': 2.540},
                                   lambda_sympy={'lambda_integral_continuity': 0.1},
                                   batch_size_per_area=128)
    self.add(ic_1, name="IntegralContinuity_1")

    # Integral Continuity 2
    ic_2 = integral_mesh.boundary_bc(outvar_sympy={'integral_continuity': -2.540},
                                     lambda_sympy={'lambda_integral_continuity': 0.1},
                                     batch_size_per_area=128)
    self.add(ic_2, name="IntegralContinuity_2")
```

Listing 118: Creating training data from STL files

14.3.2 Creating Validation and Monitor domains

The process of creating `ValidationDomain` and `MonitorDomain` is similar to previous tutorials and hence we would not cover them in detail. We will use the simulation from OpenFOAM for validating our Modulus results. Also, we will create a monitor for pressure drop across the aneurysm to monitor the convergence and compare against OpenFOAM data. The code to generate the these domains can be found below.

```
# read validation data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z', 'U:0': 'u', 'U:1': 'v', 'U:2': 'w', 'p': 'p'}
openfoam_var = csv_to_dict('openfoam/aneurysm_parabolicInlet_sol0.csv', mapping)
openfoam_invar = {key: value for key, value in openfoam_var.items() if key in ['x', 'y', 'z']}
openfoam_outvar = {key: value for key, value in openfoam_var.items() if key in ['u', 'v', 'w', 'p']}

class AneurysmVal(ValidationDomain):
  def __init__(self, **config):
    super(AneurysmVal, self).__init__()
    val = Validation.from_numpy(openfoam_invar, openfoam_outvar)
    self.add(val, name='Val')

class AneurysmMonitor(MonitorDomain):
  def __init__(self, **config):
    super(AneurysmMonitor, self).__init__()
    # metric for pressure drop
    metric = Monitor(inlet_mesh.sample_boundary(16),
                     {'pressure_drop': lambda var: tf.reduce_mean(var['p'])})
    self.add(metric, 'PressureDrop')
```

Listing 119: Defining Validation, Monitor and Inference domains

14.3.3 Making the Neural Network solver

This process is similar to other tutorials. Since we would be solving only laminar flow in this problem, we will use only `NavierStokes` and `IntegralContinuity` equations and define a network similar to tutorial 2. The code to generate the Neural Network solver is shown below.

```
class AneurysmSolver(Solver):
  train_domain = AneurysmTrain
  val_domain = AneurysmVal
  monitor_domain = AneurysmMonitor

  def __init__(self, **config):
    super(AneurysmSolver, self).__init__(**config)

    self.equations = (NavierStokes(nu=nu*scale, rho=1, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_aneurysm',
        'rec_results_cpu': True,
        'max_steps': 1500000,
        'decay_steps': 15000,
        })

if __name__ == '__main__':
  ctr = ModulusController(AneurysmSolver)
  ctr.run()
```

Listing 120: Defining the Neural Network Solver

14.4 Running the Modulus solver

Once the python file is setup, you can save the file and exit out of the text editor. In the command prompt, type in:

```
python aneurysm.py
```

14.5 Results and Post-processing

This tutorial is a good example of over-fitting the training data in the PINNs. Figure 66 shows the comparison of the validation error plots achieved for two different point densities. The case using 10 M points (total points in the .csv file) shows an initial convergence which later diverges even when the training error keeps reducing. This implies that the network is over-fitting the sampled points while sacrificing the accuracy of flow in between them. Increasing the points to 20 M solves that problem and we are able to generalize the flow field to a better resolution.

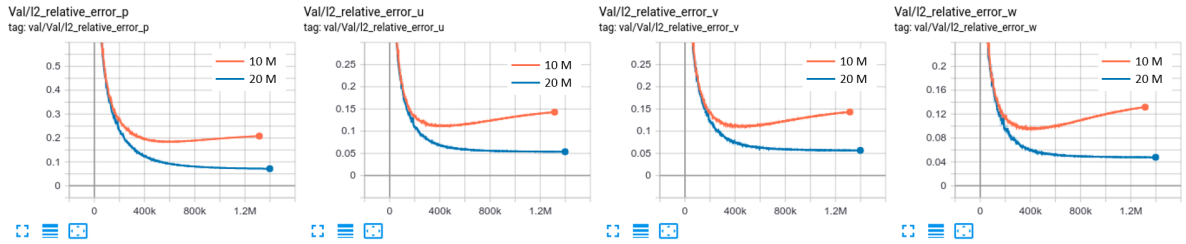


Figure 66: Convergence plots for different point density

Figure 67 shows the pressure developed inside the aneurysm and the vein. A cross-sectional view in figure 68 shows the distribution of velocity magnitude inside the aneurysm. One of the key challenges of this problem is getting the flow to develop inside the aneurysm sac and the streamline plot in figure 69 shows that Modulus successfully captures the flow field inside.

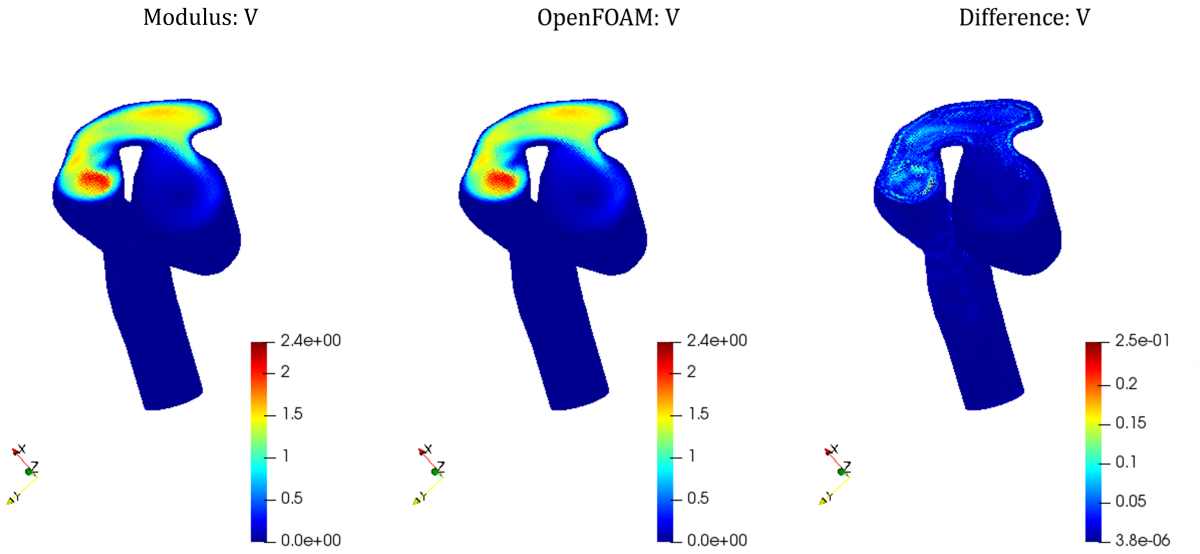


Figure 67: Cross-sectional view aneurysm showing velocity magnitude. Left: Modulus. Center: OpenFOAM. Right: Difference

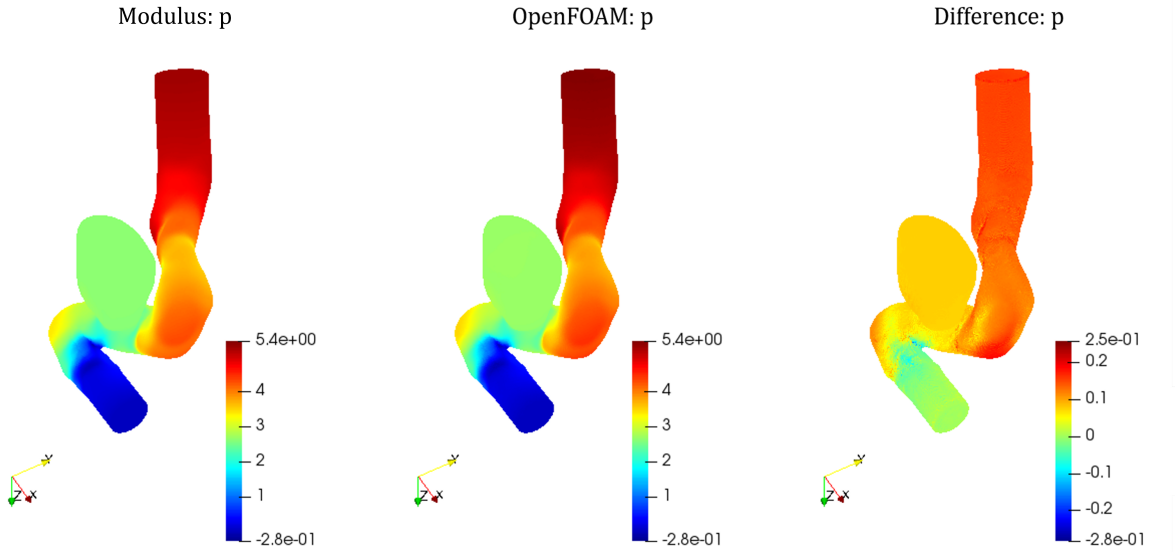


Figure 68: Pressure across aneurysm. Left: Modulus. Center: OpenFOAM. Right: Difference

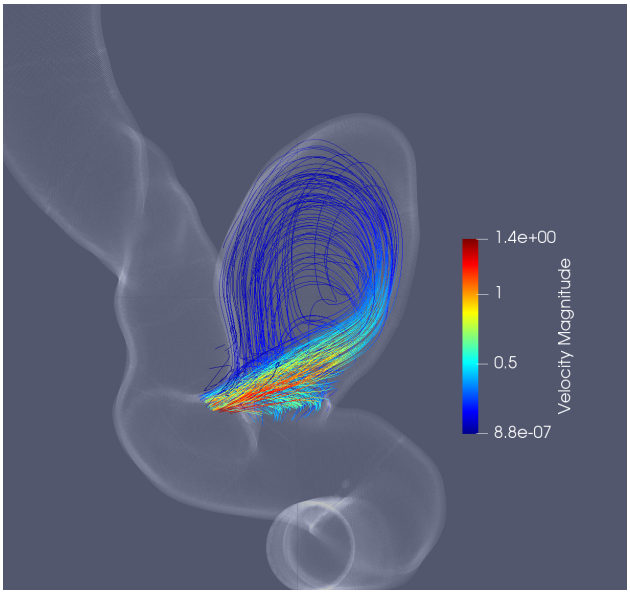


Figure 69: Flow streamlines inside the aneurysm generated from Modulus simulation.

14.6 Accelerating the Training of Neural Network Solvers via Transfer Learning

Numerous applications in science and engineering require repetitive simulations, such as simulation of blood flow in different patient-specific models. Traditional solvers simulate these models independently and from scratch. Even a minor change to the model geometry (such as an adjustment to the patient-specific medical image segmentation) requires a new simulation. Interestingly, and unlike the traditional solvers, neural network solvers can transfer knowledge across different neural network models via transfer learning. In transfer learning, the knowledge acquired by a (source) trained neural network model for a physical system is transferred to another (target) neural network model that is to be trained for a similar physical system with slightly different characteristics (such as geometrical differences). The network parameters of the target model are initialized from the source model, and are re-trained to cope with the new system characteristics without having the neural network model trained from scratch. This transfer of knowledge effectively reduces the time to convergence for neural network solvers. As an example, Figure 70 shows the application of transfer learning in training of neural network solvers for two intracranial aneurysm models with different sac shapes.

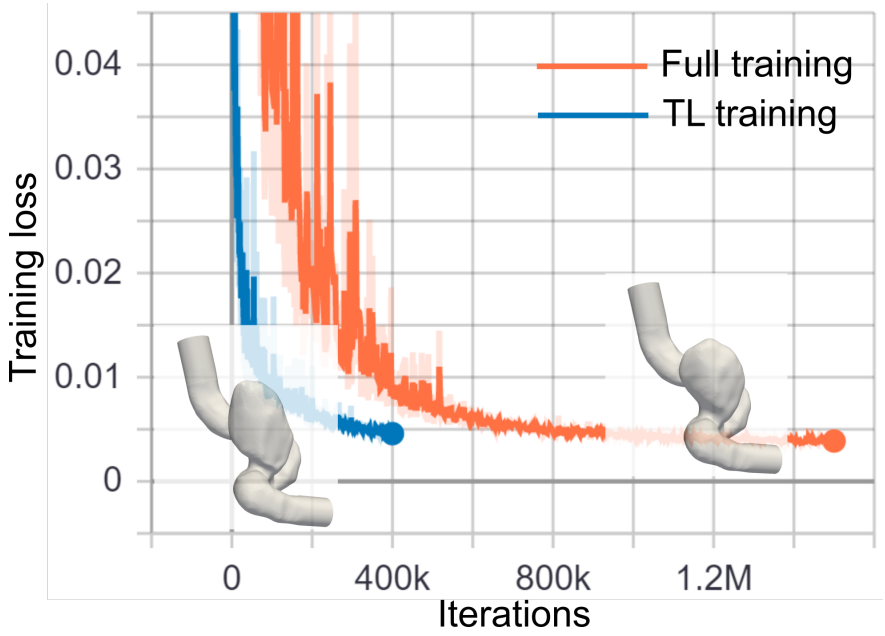


Figure 70: Transfer learning accelerates intracranial aneurysm simulations. Results are for two intracranial aneurysms with different sac shapes.

To use transfer learning in Modulus, we set `'initialize_network_dir'` to the source model network checkpoint. Also, since in transfer learning we fine-tune the source model instead of training from scratch, we use a relatively smaller learning rate compared to a full run, with smaller number of iterations and faster decay, as shown below.

```
@classmethod
def update_defaults(cls, defaults):
  defaults.update({
      'initialize_network_dir': '../aneurysm/network_checkpoint_aneurysm_source',
      'network_dir': './network_checkpoint_aneurysm_target',
      #'max_steps': 1500000,  # full run
      'max_steps': 400000,    # TL run
      #'decay_steps': 15000,  # full run
      'decay_steps': 6000,    # TL run
      #'start_lr': 1e-3,      # full run
      'start_lr': 5e-4,       # TL run
      })
```

Listing 121: Updating the training defaults for Transfer learning

15 Inverse problem: Finding unknown coefficients of a PDE

15.1 Introduction

In this tutorial, we will use Modulus to solve an inverse problem by assimilating data from observations. We will use the flow field computed by OpenFOAM as an input to the PINNs whose job would be to predict the parameters characterizing the flow (eg. viscosity (ν) and thermal diffusivity (α)). In this tutorial you would learn the following:

1. How assimilate analytical/experimental/simulation data in Modulus.
2. How to use the `BC` class in Modulus to create boundary conditions from .csv files.
3. How to use the assimilated data to make predictions of unknown quantities.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of the Modulus user interface. Lastly, we recommend you to refer to tutorial 3 for information on creating monitor and inference domains.

15.2 Problem Description

In this tutorial, we will predict the fluid's viscosity and thermal diffusivity by providing the flow field information obtained from OpenFOAM simulations as an input to the PINNs. We will use the same 2D slice from a 3-fin flow field that was used as a validation data in tutorial 8.

To summarize, the training data for this problem would be (u_i, v_i, p_i, T_i) from OpenFOAM simulation and the model would be trained to predict (ν, α) with the constraints of satisfying the governing equations of continuity, Navier-Stokes and advection-diffusion.

The ν and α used for the OpenFOAM simulation are 0.01 and 0.002 respectively.

We will scale T to define a new transport variable c for the advection-diffusion equation as shown in equation 115.

$$\begin{aligned} c &= \frac{T_{actual}}{T_{base}} - 1.0 \\ T_{base} &= 293.498K \end{aligned} \tag{115}$$

As the majority of diffusion of temperature occurs in the wake of the heat sink (Figure 71), we will only sample points in the wake for training the PINNs. We will also discard the points close to the boundary as we will be training the network to minimize loss from the conservation laws alone.

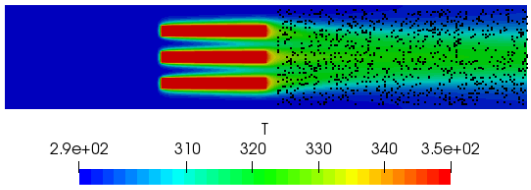


Figure 71: Batch of training points sampled from OpenFOAM data

15.3 Case Setup

In this tutorial we will use `BC` class for making the training data from .csv file. We will make three networks. First network will memorize the flow field by developing a mapping between (x, y) and (u, v, p). Second network will memorize the temperature field by developing a mapping between (x, y) and (c). A third network will be trained to invert out the desired quantities viz. (ν, α). For this problem, we will be using `NavierStokes` and `AdvectionDiffusion` equations from the PDES module.

Note: The python script for this problem can be found at `examples/three_fin_2d/heat_sink_inverse.py`.

Importing the required packages

The list of packages/modules to be imported are shown below.

```
from sympy import Symbol
import numpy as np
import tensorflow as tf

from modulus.solver import Solver
from modulus.dataset import TrainDomain, InferenceDomain, MonitorDomain
from modulus.data import BC, Monitor, Inference
from modulus.sympy_utils.geometry_2d import Rectangle, Channel2D
from modulus.csv_utils.csv_rw import csv_to_dict
from modulus.PDES.navier_stokes import NavierStokes
from modulus.PDES.advection_diffusion import AdvectionDiffusion
from modulus.controller import ModulusController
```

Listing 122: Importing the required packages and modules

15.3.1 Assimilating data from CSV files/point clouds to create Training data

`BC`

For this problem, we will not be solving the problem for the full domain. Hence, we will not be using the geometry module to create geometry for sampling points. Instead, we will use the point cloud data in form of a .csv file. We will use the `BC` class to handle such input data. The `BC` class takes in separate dictionaries for input variables and output variables. These dictionaries have a key for each variable and a numpy array of values associated to the key. Also, we will provide a batch size for sampling this .csv point cloud. This will be done by specifying the required batch size to the `batch_size` argument.

Since part of the problem involves memorizing the given flow field, we will have `['x', 'y']` as input keys and `['u', 'v', 'p', 'c', 'continuity', 'momentum_x', 'momentum_y', 'advection_diffusion']` as the output keys. Setting `['u', 'v', 'p', 'c']` as input values from OpenFOAM data, we are essentially making the network assimilate the OpenFOAM distribution of these variables in the selected domain. Setting `['continuity', 'momentum_x', 'momentum_y', 'advection_diffusion']` equal to 0, we also inform the network to satisfy the PDE losses at those sampled points. Now, except the ν and α, all the variables in these PDEs are known. Thus the network can use this information to invert out the unknowns.

The code to generate such a boundary condition is shown below.

```
# params for domain
channel_length = (-2.5, 2.5)
channel_width = (-0.5, 0.5)
heat_sink_origin = (-1, -0.3)
nr_heat_sink_fins = 3
gap = 0.15 + 0.1
heat_sink_length = 1.0
heat_sink_fin_thickness = 0.1
base_temp = 293.498

# define geometry
channel = Channel2D((channel_length[0], channel_width[0]),
                    (channel_length[1], channel_width[1]))
heat_sink = Rectangle(heat_sink_origin,
                      (heat_sink_origin[0]+heat_sink_length, heat_sink_origin[1]+heat_sink_fin_thickness))
for i in range(1, nr_heat_sink_fins):
  heat_sink_origin = (heat_sink_origin[0], heat_sink_origin[1]+gap)
  heat_sink = heat_sink + Rectangle(heat_sink_origin,
                                    (heat_sink_origin[0]+heat_sink_length, heat_sink_origin[1]+
    heat_sink_fin_thickness))
geo = channel - heat_sink

# OpenFOAM data
mapping = {'Points:0': 'x', 'Points:1': 'y',
           'U:0': 'u', 'U:1': 'v', 'p': 'p', 'T': 'c'}
openfoam_var = csv_to_dict('openfoam/heat_sink_Pr5_clipped2.csv', mapping)
openfoam_var['c'] = openfoam_var['c']/base_temp - 1.0
openfoam_invar_numpy = {key: value for key, value in openfoam_var.items()
                        if key in ['x', 'y']}
openfoam_outvar_numpy = {key: value for key, value in openfoam_var.items()
                         if key in ['u', 'v', 'p', 'c']}
openfoam_outvar_numpy['continuity'] = np.zeros_like(openfoam_outvar_numpy['u'])
openfoam_outvar_numpy['momentum_x'] = np.zeros_like(openfoam_outvar_numpy['u'])
openfoam_outvar_numpy['momentum_y'] = np.zeros_like(openfoam_outvar_numpy['u'])
```

```
openfoam_outvar_numpy['advection_diffusion']=np.zeros_like(openfoam_outvar_numpy['u'])

class HeatSinkTrain(TrainDomain):
  def __init__(self, **config):
    super(HeatSinkTrain, self).__init__()
    # interior
    interior=BC.from_numpy(openfoam_invar_numpy,openfoam_outvar_numpy,batch_size=1024)
    self.add(interior, name="Interior")
```

Listing 123: Creating training data from .csv files

Note: We have created the geometry using the geometry module only for the sake of Inference domain. The geometry generation part of the code can be omitted if an inference is not required over the entire region.

15.3.2 Creating Monitor and Inference domain

For this tutorial, we will create `MonitorDomain` to monitor the convergence of average `'nu'` and `'D'` inside the domain as the solution progresses. Once we find that the average value of these quantities have reached a steady value, we can end the simulation. The code to generate the `MonitorDomain` can be found below.

```
class HeatSinkMonitor(MonitorDomain):
  def __init__(self, **config):
    super(HeatSinkMonitor, self).__init__()
    global_monitor = Monitor(openfoam_invar_numpy, {'average_D': lambda var: tf.reduce_mean(var['D']),
                                                    'average_nu': lambda var: tf.reduce_mean(var['nu'])})
    self.add(global_monitor, 'GlobalMonitor')

class HeatSinkInference(InferenceDomain):
  def __init__(self, **config):
    super(HeatSinkInference, self).__init__()
    # save entire domain
    x, y = Symbol('x'), Symbol('y')
    interior = Inference(geo.sample_interior(1000, bounds={x: (channel_length), y: (channel_width)}),
                         ['u', 'v', 'p', 'nu', 'c', 'D'])
    self.add(interior, name="Inference")
```

Listing 124: Creating monitor domains

Note: It is also possible to create `InferenceDomain` to visualize the distribution of `'nu'` and `'D'` inside the flow field.

15.3.3 Making the Neural Network Solver for a Inverse problem

The process of creating a neural network for an inverse problem is similar to most of the problems we have seen in previous tutorials. However, as the information for the flow variables, and in turn their gradients, is already present (from OpenFOAM data) for the network to memorize, we will stop the gradient calls on each of these variables in their respective equations. This means that only the networks predicting `'nu'` and `'D'` will be optimized to minimize the equation residuals. The velocity, pressure and their gradients are treated as ground truth data.

Also, note that the the viscosity and diffusivity are passed in as Symbolic variables (`'nu'` and `'D'` respectively) to the equations as they are unknowns in this problem.

Similar to tutorial 8, we will train two separate networks to memorize flow variables (u, v, p), and scalar transport variable (c). Also, because `'nu'` and `'D'` are the custom variables we defined, we will have a separate network `'invert_net'` that produces ν and α at the output nodes.

The code to generate the neural network can be found below.

```
nu = Symbol('nu')
D = Symbol('D')

class HeatSinkSolver(Solver):
  train_domain = HeatSinkTrain
  inference_domain = HeatSinkInference
  monitor_domain = HeatSinkMonitor

  def __init__(self, **config):
    super(HeatSinkSolver, self).__init__(**config)

    self.equations = (NavierStokes(nu=nu, rho=1.0, dim=2, time=False).make_node(stop_gradients=['u', 'u__x', '
     u__x__x', 'u__y', 'u__y__y',
```

```
                                                                                                    'v', 'v__x', '
    v__x__x', 'v__y', 'v__y__y',
                                                                                                    'p', 'p__x', '
    p__y'])
                    + AdvectionDiffusion(T='c', rho=1.0, D=D, dim=2, time=False).make_node(stop_gradients=['u'
    ,
                                                                                                                'v'
    ,
                                                                                                                'c'
    , 'c__x', 'c__x__x', 'c__y', 'c__y__y']))
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    heat_net = self.arch.make_node(name='heat_net',
                                   inputs=['x', 'y'],
                                   outputs=['c'])
    invert_net = self.arch.make_node(name='invert_net',
                                     inputs=['x', 'y'],
                                     outputs=['nu', 'D'])

    self.nets = [flow_net, heat_net, invert_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_heat_sink_inverse',
        'start_lr': 5e-4,
        'max_steps': 100000,
        'decay_steps': 1000,
        })

if __name__ == '__main__':
  ctr = ModulusController(HeatSinkSolver)
  ctr.run()
```

Listing 125: Making the Modulus solver for an Inverse problem

15.4 Running the Modulus solver

Once the python file is setup, you can save the file and exit out of the text editor. In the command prompt, type in:

```
python heat_sink_inverse.py
```

15.5 Results and Post-processing

We can monitor the Tensorboard plots to see the convergence of the simulation. The Tensorboard graphs should look similar to the ones shown in figure 72.

Table 5: Comparison of the inverted coefficients with the actual values

Property	OpenFOAM (True)	Modulus (Predicted)
Kinematic Viscosity (m^2/s)	1.00×10^{-2}	1.03×10^{-2}
Thermal Diffusivity (m^2/s)	2.00×10^{-3}	2.19×10^{-3}

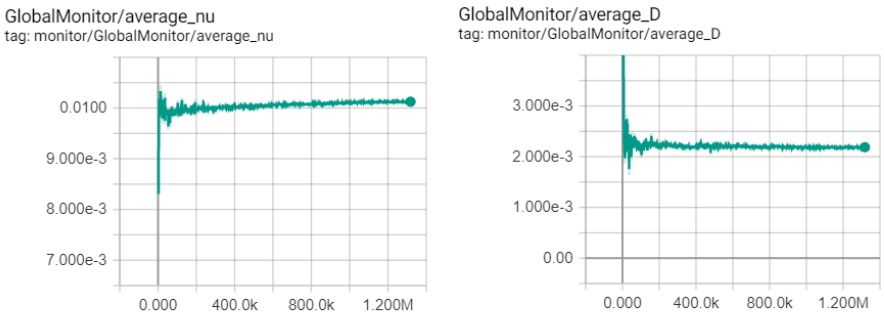


Figure 72: Tensorboard plots for ν and α

16 Parameterized Simulations and Design Optimization: 3D heat sink

16.1 Introduction

In this tutorial, we will walk through the process of simulating a parameterized problem using Modulus. The neural networks in Modulus allow us to solve problems for multiple design parameters in a single training. This implies that once the training is complete, it is possible to run inference on several geometry/physical parameter combinations as a post-processing step, without solving the forward problem again. We will see that such parameterization increases the computational cost only fractionally while solving the entire desired design space.

To demonstrate this feature we will solve the flow and heat over a 3-fin heat sink whose fin height, fin thickness, and fin length are variable. We will then perform a design optimization to find out the most optimal fin configuration for our heat sink example. By the end of this tutorial, you would learn to easily convert any simulation to a parametric study using Modulus' Geometry module and Neural Network solver, and also perform design optimization. In this tutorial, you would learn the following:

1. How to set up a parametric simulation in Modulus.
2. How to perform design optimization.

Prerequisites

This tutorial is an extension of tutorial 13 where we discussed how to use Modulus for solving Conjugate Heat problems. In this tutorial, we take up the same geometry setup and solve it for a parameterized setup at an increased Reynolds number. Hence, we recommend you to refer tutorial 13 for any additional details related to geometry specification and boundary conditions.

Note: In this tutorial we will focus on parameterization which is independent of the physics being solved and can be applied to any class of problems covered in the User Guide.

16.2 Problem Description

Please refer the geometry and boundary conditions for a 3-fin heat sink in tutorial 13. We will parameterize this problem to solve for several heat sink designs in a single neural network training. We will modify the heat sink's fin dimensions (thickness, length and height) to create a design space of various heat sinks. In this tutorial, we change the Reynolds number of 50 in tutorial 13 and increase it to 500 and incorporate turbulence using Zero Equation turbulence model.

More accurately, for this problem, we will vary the height (h), length (l), and thickness (t) of the central fin and the two side fins. The height, length, and thickness of the two side fins are kept the same, and therefore, there will be a total of six geometry parameters. The ranges of variation for these geometry parameters are given in equation 116.

$$\begin{aligned} h_{centralfin} &= (0.0, 0.6), \\ h_{sidefins} &= (0.0, 0.6), \\ l_{centralfin} &= (0.5, 1.0) \\ l_{sidefins} &= (0.5, 1.0) \\ t_{centralfin} &= (0.05, 0.15) \\ t_{sidefins} &= (0.05, 0.15) \end{aligned} \tag{116}$$

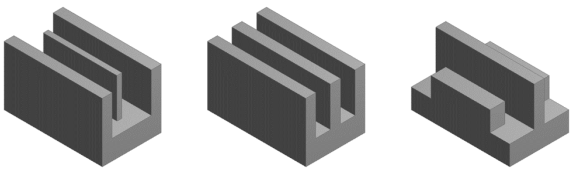


Figure 73: Examples of some of the 3 Fin geometries covered in the chosen design space

16.3 Case Setup

In this tutorial, we will use the 3D geometry module from Modulus to create the parameterized 3-fin heat sink geometry. The current version of Modulus does not support the parameterization of some of the discrete features like the number of fins. Hence in this tutorial we will only cover parameters that are continuous. Also, we will be training the parameterized model and validating it by performing inference on a case where $h_{centralfin} = 0.4$, $h_{sidefins} = 0.4$, $l_{centralfin} = 1.0$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.1$, and $t_{sidefins} = 0.1$. At the end of the tutorial we will also present the comparison between results for the above combination of parameters obtained from a parameterized model versus results obtained from a non-parameterized model trained on just a single geometry corresponding to the same set of values. This will highlight the usefulness of using PINNs for doing parameterized simulations in comparison to some of the traditional methods.

As we did in the tutorial 13, we will split the code in three parts, the domain, flow solver and heat solver. The design optimization would be run as a post-processing step after both the flow and heat are trained using a separate script.

Hence, we will split the code into the following different parts:

1. `three_fin_domain_zero_eq_parameterized.py`: This file will contain all the definitions of geometry, boundary conditions, equations, and definition of validation, monitor, and inference domains.
2. `three_fin_flow_solver_zero_eq_parameterized.py`: This file will contain the solver details for flow. It will derive all the required information from the `three_fin_domain_zero_eq_parameterized.py` file which would be imported at the beginning of the code.
3. `three_fin_heat_solver_zero_eq_parameterized.py`: This file will contain the solver details for heat. It will derive all the required information from the `three_fin_domain_zero_eq_parameterized.py` file which would be imported at the beginning of the code.
4. `three_fin_optimizer_zero_eq_parameterized.py`: This file will contain the brute force optimization details and it uses the trained checkpoints generated by the solver files above.

Note: All the files for this problem can be found at `examples/three_fin_3d/turbulent_parameterized_with_design_optimization/`.

Importing the required packages

The list of packages/modules to be imported in each of the file are similar to tutorial 13. So we will skip the description here. Only additional packages that we would import for this tutorial are `ZeroEquation` from `PDES` module for turbulence. The detailed packages imported can be found in the scripts discussed in the previous section.

16.3.1 Creating the Parameterized Geometry

As described earlier, we will edit the `three_fin_domain_zero_eq_parameterized.py` file to create the geometry. We will use all the same primitives that we used in the tutorial 13.

To create the parameteric shapes and solve them for any problem in Modulus, there are 4 things that we need to in addition to a regular single geometry problem.

1. Create symbolic variables for the dimensions we want to parameterize
2. Define ranges of variation for all the variables created in step 1
3. Generate geometry using Modulus' geometry module by supplying the symbolic variables as inputs
4. Pass the parameteric ranges to the `param_ranges` attribute of `interior_bc` and `boundary_bc`.

Then, any symbolic parameters that we may have created (geometric or non geometric) need to be passed as inputs in addition to the Cartesian coordinates while making the neural network in the solver file.

Hence, for this problem, we pass in the symbolic variables for the geometric dimensions. We will create symbolic variables for length, thickness and height for the side fins and center/middle fins separately using sympy `Symbol`.

The code to generate the parameterized geometry can be found below (Note: In the scripts, the fin height for is measured from the top of the heat sink base):

```
# define sympy varaibles to parametize domain curves
x, y, z = Symbol('x'), Symbol('y'), Symbol('z')
```

```
# parametric variation
fin_height_m, fin_height_s = Symbol('fin_height_m'), Symbol('fin_height_s')
fin_length_m, fin_length_s = Symbol('fin_length_m'), Symbol('fin_length_s')
fin_thickness_m, fin_thickness_s = Symbol('fin_thickness_m'), Symbol('fin_thickness_s')
height_m_range = (0.0, 0.6)
height_s_range = (0.0, 0.6)
length_m_range = (0.5, 1.0)
length_s_range = (0.5, 1.0)
thickness_m_range = (0.05, 0.15)
thickness_s_range = (0.05, 0.15)
param_ranges = {fin_height_m: height_m_range,
                fin_height_s: height_s_range,
                fin_length_m: length_m_range,
                fin_length_s: length_s_range,
                fin_thickness_m: thickness_m_range,
                fin_thickness_s: thickness_s_range}

# geometry params for domain
channel_origin        = (-2.5, -0.5, -0.5)
channel_dim           = (5.0, 1.0, 1.0)
heat_sink_base_origin = (-1.0, -0.5, -0.3)
heat_sink_base_dim    = (1.0, 0.2, 0.6)
fin_origin            = (heat_sink_base_origin[0]+0.5-fin_length_s/2, -0.3, -0.3)
fin_dim               = (fin_length_s, fin_height_s, fin_thickness_s)       # two side fins
total_fins            = 2                                                   # two side fins
flow_box_origin       = (-1.1, -0.5, -0.5)
flow_box_dim          = (1.6, 1.0, 1.0)
source_origin         = (-0.7, -0.5, -0.1)
source_dim            = (0.4, 0.0, 0.2)
source_area           = 0.08

# params for simulation
# fluid params
nu                    = 0.002
rho                   = 1
inlet_vel             = 1.0
volumetric_flow       = 1.0
# heat params
D_solid               = 0.0625
D_fluid               = 0.02
inlet_t               = 293.15
grad_t                = 360
inlet_t = inlet_t/273.15 - 1.0
grad_t = grad_t/273.15

# define sympy varaibles to parametize domain curves
x, y, z = Symbol('x'), Symbol('y'), Symbol('z')

# define geometry
# channel
channel = Channel(channel_origin, (channel_origin[0]+channel_dim[0],
                                   channel_origin[1]+channel_dim[1],
                                   channel_origin[2]+channel_dim[2]))

# three fin heat sink
heat_sink_base = Box(heat_sink_base_origin, (heat_sink_base_origin[0]+heat_sink_base_dim[0], # base of heat
    sink
                                             heat_sink_base_origin[1]+heat_sink_base_dim[1],
                                             heat_sink_base_origin[2]+heat_sink_base_dim[2]))
fin_center = (fin_origin[0] + fin_dim[0]/2, fin_origin[1] + fin_dim[1]/2, fin_origin[2] + fin_dim[2]/2)
fin = Box(fin_origin, (fin_origin[0]+fin_dim[0],
                       fin_origin[1]+fin_dim[1],
                       fin_origin[2]+fin_dim[2]))
gap = (heat_sink_base_dim[2]-fin_dim[2])/(total_fins-1) # gap between fins
fin.repeat(gap, repeat_lower=(0,0,0), repeat_higher=(0,0,total_fins-1), center=fin_center)
three_fin = heat_sink_base + fin

# parameterized center fin
center_fin_origin = (heat_sink_base_origin[0]+0.5-fin_length_m/2, fin_origin[1], -fin_thickness_m/2)
center_fin_dim = (fin_length_m, fin_height_m, fin_thickness_m)
center_fin = Box(center_fin_origin, (center_fin_origin[0]+center_fin_dim[0],
                                     center_fin_origin[1]+center_fin_dim[1],
                                     center_fin_origin[2]+center_fin_dim[2]))
three_fin = three_fin + center_fin

# entire geometry
geo = channel - three_fin

# low and high resolution geo away and near the heat sink
flow_box = Box(flow_box_origin, (flow_box_origin[0]+flow_box_dim[0], # base of heat sink
                                 flow_box_origin[1]+flow_box_dim[1],
```

```
                                flow_box_origin[2]+flow_box_dim[2]))
lr_geo = geo - flow_box
hr_geo = geo & flow_box
lr_bounds_x = (channel_origin[0], channel_origin[0] + channel_dim[0])
lr_bounds_y = (channel_origin[1], channel_origin[1] + channel_dim[1])
lr_bounds_z = (channel_origin[2], channel_origin[2] + channel_dim[2])
hr_bounds_x = (flow_box_origin[0], flow_box_origin[0] + flow_box_dim[0])
hr_bounds_y = (flow_box_origin[1], flow_box_origin[1] + flow_box_dim[1])
hr_bounds_z = (flow_box_origin[2], flow_box_origin[2] + flow_box_dim[2])

# inlet and outlet
inlet = Plane(channel_origin, (channel_origin[0], channel_origin[1]+channel_dim[1], channel_origin[2]+
    channel_dim[2]), -1)
outlet = Plane((channel_origin[0]+channel_dim[0], channel_origin[1], channel_origin[2]), (channel_origin[0]+
    channel_dim[0], channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]), 1)

# planes for integral continuity
x_pos = Symbol('x_pos')
integral_plane = Plane((x_pos, channel_origin[1], channel_origin[2]),
                       (x_pos, channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]),
                       1)
x_pos_range = {x_pos: lambda batch_size: np.full((batch_size, 1), np.random.uniform(-1.1, 0.1))}
fixed_param_range = {fin_height_m:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0, 0.6)),
                     fin_height_s:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0, 0.6)),
                     fin_length_m:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.5, 1.0)),
                     fin_length_s:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.5, 1.0)),
                     fin_thickness_m: lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.05, 0.15))
    ,
                     fin_thickness_s: lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.05, 0.15))
    }
```

Listing 126: Generating the parameterized geometry

16.3.2 Defining the Boundary conditions and Equations for a parameterized problem

The boundary conditions for the flow and heat are similar to the tutorial 13, so we will again skip the description of them here. Key thing to note for parameterized problems is that, we need to input the `param_ranges=param_ranges` to each of the boundary conditions (`boundary_bc`, `interior_bc`) created to ensure we are solving the parameterized problem. For the parameterized integral continuity planes, we would combine the fixed parameter dictionary with the x position dictionary of IC planes using `{**x_pos_range, **fixed_param_range}` dictionary.

Note: For simplicity, we just show the flow boundary conditions and equations for a parameterized problem here. Similar steps can be taken to defined the heat boundary conditions. The detailed script can be referred in the `examples/` directory for the exact definition of each boundary condition and equation.

```
# define flow domain
class ThreeFinFlowTrain(TrainDomain):
  def __init__(self, **config):
    super(ThreeFinFlowTrain, self).__init__()
    # inlet
    inletBC = inlet.boundary_bc(outvar_sympy={'u': inlet_vel, 'v': 0, 'w': 0},
                                batch_size_per_area=500,
                                lambda_sympy={'lambda_u': channel.sdf, # weight zero on edges
                                              'lambda_v': 1.0,
                                              'lambda_w': 1.0},
                                criteria=Eq(x, channel_origin[0]),
                                param_ranges=param_ranges)
    self.add(inletBC, name="Inlet")

    # outlet
    outletBC = outlet.boundary_bc(outvar_sympy={'p': 0},
                              batch_size_per_area=500,
                              criteria=Eq(x, channel_origin[0]+channel_dim[0]),
                              param_ranges=param_ranges)
    self.add(outletBC, name="Outlet")

    # no slip for channel walls
    no_slip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                              batch_size_per_area=500,
                              param_ranges=param_ranges)
    self.add(no_slip, name="NoSlipChannel")

    # flow interior low res away from three fin
    interiorF_lr = lr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                      bounds={x: lr_bounds_x, y: lr_bounds_y, z: lr_bounds_z},
```

```
                                    lambda_sympy={'lambda_continuity': geo.sdf,
                                                  'lambda_momentum_x': geo.sdf,
                                                  'lambda_momentum_y': geo.sdf,
                                                  'lambda_momentum_z': geo.sdf},
                                    batch_size_per_area=500,
                                    param_ranges=param_ranges)
    self.add(interiorF_lr, name="FlowInterior_LR")

    # flow interiror high res near three fin
    interiorF_hr = hr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                    bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                    lambda_sympy={'lambda_continuity': geo.sdf,
                                                  'lambda_momentum_x': geo.sdf,
                                                  'lambda_momentum_y': geo.sdf,
                                                  'lambda_momentum_z': geo.sdf},
                                    batch_size_per_area=3000,
                                    param_ranges=param_ranges)
    self.add(interiorF_hr, name="FlowInterior_HR")

    # integral continuity
    for i in range(5):
      IC = integral_plane.boundary_bc(outvar_sympy={'integral_continuity': 1.0},
                                    batch_size_per_area=10000,
                                    lambda_sympy={'lambda_integral_continuity': 1.0},
                                    criteria=geo.sdf>0,
                                    param_ranges={**x_pos_range, **fixed_param_range},
                                    fixed_var=False)
      self.add(IC, name="IntegralContinuity_"+str(i))
```

Listing 127: Defining the flow boundary conditions and equations to solve

16.3.3 Creating Validation, Monitor and Inference domain for a parameterized simulation

We will use the validation data from OpenFOAM that was generated for the following combination of parameters: $h_{centralfin} = 0.4$, $h_{sidefins} = 0.4$, $l_{centralfin} = 1.0$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.1$, and $t_{sidefins} = 0.1$. This will be done by updating the .csv file from the OpenFOAM simulation with the above values for the newly defined parameters. The code to do the same can be found below.

Again, for simplicity, we only show the validation domain for the flow field's flow part. Similar process needs to be followed for the validation domain of the heat part where we include both fluid and solid and the details can be referred in the actual scripts.

```
# validation data
# flow data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z',
           'U:0': 'u', 'U:1': 'v','U:2': 'w', 'p_rgh': 'p', 'T': 'theta_f'}
openfoam_var = csv_to_dict('../openfoam/threeFin_extend_zeroEq_re500_fluid.csv', mapping)
openfoam_var['theta_f'] = openfoam_var['theta_f']/273.15 - 1.0 # normalize heat
openfoam_var['x'] = openfoam_var['x'] + channel_origin[0]
openfoam_var['y'] = openfoam_var['y'] + channel_origin[1]
openfoam_var['z'] = openfoam_var['z'] + channel_origin[2]
openfoam_var.update({'fin_height_m': np.full_like(openfoam_var['x'], 0.4)})
openfoam_var.update({'fin_height_s': np.full_like(openfoam_var['x'], 0.4)})
openfoam_var.update({'fin_length_m': np.full_like(openfoam_var['x'], 1.0)})
openfoam_var.update({'fin_length_s': np.full_like(openfoam_var['x'], 1.0)})
openfoam_var.update({'fin_thickness_m': np.full_like(openfoam_var['x'], 0.1)})
openfoam_var.update({'fin_thickness_s': np.full_like(openfoam_var['x'], 0.1)})
openfoam_invar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['x','y','z','
     fin_height_m','fin_length_m','fin_thickness_m','fin_height_s','fin_length_s','fin_thickness_s']}
openfoam_outvar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['u','v','w','p']}
openfoam_outvar_flow_heat_numpy = {key: value for key, value in openfoam_var.items() if key in ['theta_f']}

class ThreeFinFlowVal(ValidationDomain):
  def __init__(self, **config):
    super(ThreeFinFlowVal, self).__init__()
    # fluid validation
    val = Validation.from_numpy(openfoam_invar_flow_numpy, openfoam_outvar_flow_numpy)
    self.add(val, name='ValFlowField')
```

Listing 128: Defining the validation domain for a parametric simulation

For this problem, for the flow part, we monitored the residuals of the continuity and momentum equations along with pressure drop across the heat sink in tutorial 13. The process to define the `MonitorDomain` domains is similar, but as

an addition, for parametric simulations, we will have to supply the values for the parameters form which we would like to monitor these values. Like validation data, we will monitor these values for the following combination of parameters: $h_{centralfin} = 0.6$, $h_{sidefins} = 0.6$, $l_{centralfin} = 1.0$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.1$, and $t_{sidefins} = 0.1$. The definition of the heat domains can be referred in the scripts.

```
class ThreeFinFlowMonitor(MonitorDomain):
  def __init__(self, **config):
    super(ThreeFinFlowMonitor, self).__init__()
    # metric for equation residuals
    global_monitor = Monitor(geo.sample_interior(10000, bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
      param_ranges={fin_height_m: 0.4, fin_height_s: 0.4, fin_length_m: 1.0, fin_length_s: 1.0, fin_thickness_m:
      0.1, fin_thickness_s: 0.1}),
                              {'mass_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['continuity'])),

                              'momentum_x_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_x'])),
                              'momentum_y_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_y'])),
                              'momentum_z_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_z']))})
    self.add(global_monitor, 'ResiduallMonitor')

    # metric for pressure drop front
    p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges={x_pos: heat_sink_base_origin[0]-
     heat_sink_base_dim[0], fin_height_m: 0.4, fin_height_s: 0.4, fin_length_m: 1.0, fin_length_s: 1.0,
     fin_thickness_m: 0.1, fin_thickness_s: 0.1}),
                    {'pressure': lambda var: tf.reduce_mean(var['p'])})
    self.add(p_monitor, 'FrontPressure')
    p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges={x_pos: heat_sink_base_origin[0]+2*
     heat_sink_base_dim[0], fin_height_m: 0.4, fin_height_s: 0.4, fin_length_m: 1.0, fin_length_s: 1.0,
     fin_thickness_m: 0.1, fin_thickness_s: 0.1}),
                    {'pressure': lambda var: tf.reduce_mean(var['p'])})
    self.add(p_monitor, 'BackPressure')
```

Listing 129: Defining the monitor domain for a parameteric simulation

16.3.4 Making the Neural Network Solver for a parameterized problem

Once all the parameterized domain definitions are completed, for training the parameterized model, we will have the symbolic parameters we defined earlier as inputs to the neural network architecture in `flow_net` viz. `'fin_height_m'`, `'fin_height_s'`, `'fin_thickness_m'`, `'fin_thickness_s'`, `'fin_length_m'`, `'fin_length_s'` along with the usual x, y, z coordinates. The outputs remain the same as what we would have for any other non-parameterized simulation. The below code shows the flow solver file for the paramterized problem.

```
class ThreeFinFlowSolver(Solver):
  train_domain = ThreeFinFlowTrain
  monitor_domain = ThreeFinFlowMonitor
  val_domain = ThreeFinFlowVal

  def __init__(self, **config):
    super(ThreeFinFlowSolver, self).__init__(**config)
    self.equations = (NavierStokes(nu='nu', rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node()
                      + ZeroEquation(nu=nu, dim=3, time=False, max_distance=0.5).make_node()
                      + [Node.from_sympy(geo.sdf, 'normal_distance')])
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z',
                                   'fin_height_m', 'fin_height_s',
                                   'fin_length_m', 'fin_length_s',
                                   'fin_thickness_m', 'fin_thickness_s'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_three_fin_fluid_flow_parameterized_zero_eq',
        'rec_results_cpu': True,
        'max_steps': 1500000,
        'start_lr': 5e-4
        })

if __name__ == '__main__':
  ctr = ModulusController(ThreeFinFlowSolver)
  ctr.run()
```

Listing 130: Making the Modulus solver for parameterized simulations

16.4 Running the Modulus solver

This part is exactly similar to tutorial 13 and once all the definitions are complete, we can execute the parameterized problem like any other problem. The sequence in which the scripts need to be executed can be summarized as follows:

1. Solve the flow equations first (run `three_fin_flow_solver_zero_eq_parameterized.py`)
2. After convergence of flow, solve the heat equations (run `three_fin_heat_solver_zero_eq_parameterized.py`)
3. Optional design optimization: After convergence of heat, run inference for finding the optimal design (run `three_fin_optimizer_zero_eq_parameterized.py`)

16.5 Design Optimization

As we discussed previously, we can optimize the design once the training is complete by making only some minor modifications. A typical design optimization usually contains an objective function that we intend to minimize/maximize subject to some physical/design constraints.

For heat sink designs, usually the peak temperature that can be reached at the source chip is limited. This limit arises from the operating temperature requirements of the chip on which the heat sink is mounted for cooling purposes. The design is then constrained by the maximum pressure drop that can be successfully provided by the cooling system that pushes the flow around the heat sink. Mathematically this can be expressed as below:

Table 6: Optimization problem

	Variable/Function	Description
minimize	$Peak\ Temperature$	Minimize the peak temperature at the source chip
with respect to	$h_{centralfin}, h_{sidefins}$, $l_{centralfin}$, $l_{sidefins}$, $t_{centralfin}, t_{sidefins}$	Geometric Design Variables of the Heat Sink
subject to	$Pressure\ drop < 2.5$	Limit on the pressure drop. Maximum value of the pressure drop that can be provided by the cooling system

Let's now see how to tackle this problem using Modulus.

As you have noticed, while solving the parameterized simulation we created some monitors to track the peak temperature and the pressure drop for some design variable combination. We basically would follow the same process and use the `Monitor` domain to find the values for multiple combinations of the design variables this time. We can create this simply by looping through the multiple designs. Since we would be looping through several variable combinations, we recommend to add these monitors only after the training is complete to achieve better computational efficiency.

The following snippet of code needs to be added to the domain file (`three_fin_domain_zero_eq_parameterized.py`) to be able to sample the pressure and temperature values for the required designs.

```
class ThreeFinDesignOptMonitor(MonitorDomain):
  def __init__(self, **config):
    super(ThreeFinDesignOptMonitor, self).__init__()

    # define candidate designs
    num_samples = 3
    inference_param_tuple  = itertools.product(np.linspace(*height_m_range, num_samples),
                                               np.linspace(*height_s_range, num_samples),
                                               np.linspace(*length_m_range, num_samples),
                                               np.linspace(*length_s_range, num_samples),
                                               np.linspace(*thickness_m_range, num_samples),
```

```
                                            np.linspace(*thickness_s_range, num_samples))
    for (HS_height_m_, HS_height_s_, HS_length_m_, HS_length_s_, HS_thickness_m_, HS_thickness_s_) in
     inference_param_tuple:
      HS_height_m = float(HS_height_m_)
      HS_height_s = float(HS_height_s_)
      HS_length_m = float(HS_length_m_)
      HS_length_s = float(HS_length_s_)
      HS_thickness_m = float(HS_thickness_m_)
      HS_thickness_s = float(HS_thickness_s_)
      specific_param_ranges = {fin_height_m: HS_height_m,
                               fin_height_s: HS_height_s,
                               fin_length_m: HS_length_m,
                               fin_length_s: HS_length_s,
                               fin_thickness_m: HS_thickness_m,
                               fin_thickness_s: HS_thickness_s}

      # add metrics for pressure drop
      plane_param_ranges = {**specific_param_ranges, **{x_pos: heat_sink_base_origin[0]-heat_sink_base_dim[0]}}
      p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges=plane_param_ranges),
                          {'pressure': lambda var: tf.reduce_mean(var['p'])})
      self.add(p_monitor, 'FP_'+str(HS_height_m)+'_'+str(HS_height_s)+'_'+str(HS_length_m)+'_'
                               +str(HS_length_s)+'_'+str(HS_thickness_m)+'_'+str(HS_thickness_s))

      plane_param_ranges = {**specific_param_ranges, **{x_pos: heat_sink_base_origin[0]+2*heat_sink_base_dim
     [0]}}
      p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges=plane_param_ranges),
                          {'pressure': lambda var: tf.reduce_mean(var['p'])})
      self.add(p_monitor, 'BP_'+str(HS_height_m)+'_'+str(HS_height_s)+'_'+str(HS_length_m)+'_'
                               +str(HS_length_s)+'_'+str(HS_thickness_m)+'_'+str(HS_thickness_s))

      # add metrics for peak temp
      plane_param_ranges = {**specific_param_ranges}
      temp_monitor = Monitor(three_fin.sample_boundary(10000, criteria=Eq(y, source_origin[1]), param_ranges=
     plane_param_ranges),
                         {'peak_temp': lambda var: tf.reduce_max(var['theta_s'])})
      self.add(temp_monitor, 'PT_'+str(HS_height_m)+'_'+str(HS_height_s)+'_'+str(HS_length_m)+'_'
                               +str(HS_length_s)+'_'+str(HS_thickness_m)+'_'+str(HS_thickness_s))
```

Listing 131: Making the Monitors for sampling multiple design configurations

Once the monitors are defined in the domain file, all we need to do is run the solver files in the `'eval'` mode to be able to produce inference on all the sampled variable combinations.

This can be achieved using the flag `--run_mode=eval` while executing the solver python scripts again after the initial training. This will populate the `/network_checkpoint_.../monitor_domain/results/` directory with all the designs and their corresponding peak temperature and pressure drops for us to look at. We can then select the design with the least peak temperature and the maximum pressure drop < 2.5.

Doing this process manually is tedious as it requires going through several csv files and then finding the optimal. Alternatively, the same thing can be written in a script. Below we show the contents of `three_fin_optimizer_zero_eq_parameterized.py`. This file basically reads in the new monitor domain we defined, executes the required scripts in `'eval'` mode and then ranks the designs in the order of most optimal to least optimal under the specified constraints.

```
# import domain
from three_fin_domain_zero_eq_parameterized import ThreeFinHeatTrain, ThreeFinDesignOptMonitor
from three_fin_heat_solver_zero_eq_parameterized import ThreeFinHeatSolver

# import Modulus library
from modulus.solver import Solver
from modulus.controller import ModulusController
from modulus.csv_utils.csv_rw import dict_to_csv

# import other libraries
import numpy as np
import os
import csv

# specify the design optimization requirements
max_pressure_drop = 2.5
num_design = 10
path = './network_checkpoint_three_fin_heat_parameterized_zero_eq'
invar_mapping = ['fin_height_middle',
                 'fin_height_sides',
                 'fin_length_middle',
                 'fin_length_sides',
```

```
                'fin_thickness_middle',
                'fin_thickness_sides']
outvar_mapping = ['pressure_drop', 'peak_temp']

# run the monitor domain for front and back pressures and the peak temp
class ThreeFinDesignOpt(Solver):
  train_domain = ThreeFinHeatTrain
  monitor_domain = ThreeFinDesignOptMonitor

  def __init__(self, **config):
    super(ThreeFinDesignOpt, self).__init__(**config)
    heat_solver = ThreeFinHeatSolver(**config)
    self.equations = heat_solver.equations
    self.nets = heat_solver.nets

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'initialize_network_dir': './network_checkpoint_three_fin_fluid_flow_parameterized_zero_eq',
        'network_dir': './network_checkpoint_three_fin_heat_parameterized_zero_eq',
        'rec_results_cpu': True,
        'run_mode': 'eval'
        })

# read the monitor files, and perform a brute force design optimization
def DesignOpt(path, num_design, max_pressure_drop, invar_mapping, outvar_mapping):
  path += '/monitor_domain/results'
  directory = os.path.join(os.getcwd(), path)
  sys.path.append(path)
  values = []
  configs=[]
  for _, _, files in os.walk(directory):
      for file in files:
          if file.startswith("BP") & file.endswith(".csv"):
              value = []
              configs.append(file[2:-4])

              # read back pressure
              with open(os.path.join(path, file), "r") as datafile:
                  data=[]
                  reader = csv.reader(datafile, delimiter=',')
                  for row in reader:
                      columns = [row[1]]
                      data.append(columns)
                  last_row = float(data[-1][0])
                  value.append(last_row)

              # read front pressure
              with open(os.path.join(path,'FP'+file[2:]), "r") as datafile:
                  reader = csv.reader(datafile, delimiter=',')
                  data=[]
                  for row in reader:
                      columns = [row[1]]
                      data.append(columns)
                  last_row = float(data[-1][0])
                  value.append(last_row)

              # read temperature
              with open(os.path.join(path,'PT'+file[2:]), "r") as datafile:
                  data=[]
                  reader = csv.reader(datafile, delimiter=',')
                  for row in reader:
                      columns = [row[1]]
                      data.append(columns)
                  last_row = float(data[-1][0])
                  value.append(last_row)
              values.append(value)

  # perform the design optimization
  values = np.array([[values[i][1]-values[i][0], values[i][2]*273.15] for i in range(len(values))])
  indices = np.where(values[:,0]<max_pressure_drop)[0]
  values = values[indices]
  configs = [configs[i] for i in indices]
  opt_design_index = values[:,1].argsort()[0:num_design]
  opt_design_values = values[opt_design_index]
  opt_design_configs = [configs[i] for i in opt_design_index]

  # Save to a csv file
  opt_design_configs = np.array([np.array(opt_design_configs[i][1:].split('_')).astype(float) for i in range(
     num_design)])
```

```
  opt_design_configs_dict = {key: value.reshape(-1,1) for (key,value) in zip(invar_mapping, opt_design_configs.
     T)}
  opt_design_values_dict = {key: value.reshape(-1,1) for (key,value) in zip(outvar_mapping, opt_design_values.T)
     }
  opt_design = {**opt_design_configs_dict, **opt_design_values_dict}
  dict_to_csv(opt_design,'optimal_design')
  print('Finished design optimization!')

if __name__ == '__main__':
  ctr = ModulusController(ThreeFinDesignOpt)
  ctr.run()
  DesignOpt(path, num_design, max_pressure_drop, invar_mapping, outvar_mapping)
```

Listing 132: The Optimizer Script

The above script can then be run using by executing the following command.

```
python three_fin_optimizer_zero_eq_parameterized.py
```

16.6 Results and Post-processing

The output of the optimization script is stored in a .csv file which ranks 20 designs from the sample with the most optimal desing ranked first. The design parameters for the optimal heat sink for this problem are: $h_{centralfin} = 0.4$, $h_{sidefins} = 0.4$, $l_{centralfin} = 0.83$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.15$, $t_{sidefins} = 0.15$. The above design has a pressure drop of 2.46 and a peak temperature of 76.23 $°C$ (Figure 74)

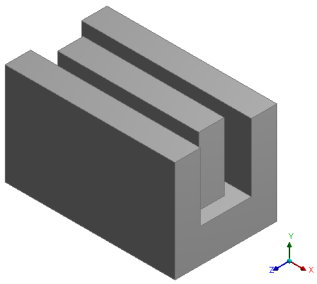


Figure 74: Three Fin geometry after optimization

Table 7 represents the computed pressure drop and peak temperature for the OpenFOAM single geometry and Modulus single and parameterized geometry runs. It is evident that the results for the parameterized model are close to those of a single geometry model, showing its good accuracy.

Table 7: A comparison for the OpenFOAM and Modulus results

Property	OpenFOAM Single Run	Single Run	Parameterized Run
Pressure Drop (Pa)	2.195	2.063	2.016
Peak Temperature $(°C)$	72.68	76.10	77.41

By parameterizing the geometry, Modulus significantly accelerates design optimization when compared to traditional solvers, which are limited to single geometry simulations. For instance, 3 values (two end values of the range and a middle value) per design variable would result in 3^6 = 729 single geometry runs. The total compute time required by OpenFOAM and Modulus for this design optimization task is reported in Table 8. It can be seen that for 729 OpenFOAM runs, it would result in 4099 wall-hrs of compute time while the neural network takes only 120 wall-hrs of compute time. Large number of design variables or their values would only magnify the difference in the time taken for two approaches.

Table 8: Total compute time needed for OpenFOAM and Modulus for the three fin heat sink design optimization

Solver	OpenFOAM	Modulus
Compute Time (hrs)	4099	120

Note: The Modulus calculations were done using 4 Nvidia V100 GPUs. The OpenFOAM calculations were done using 20 processors.

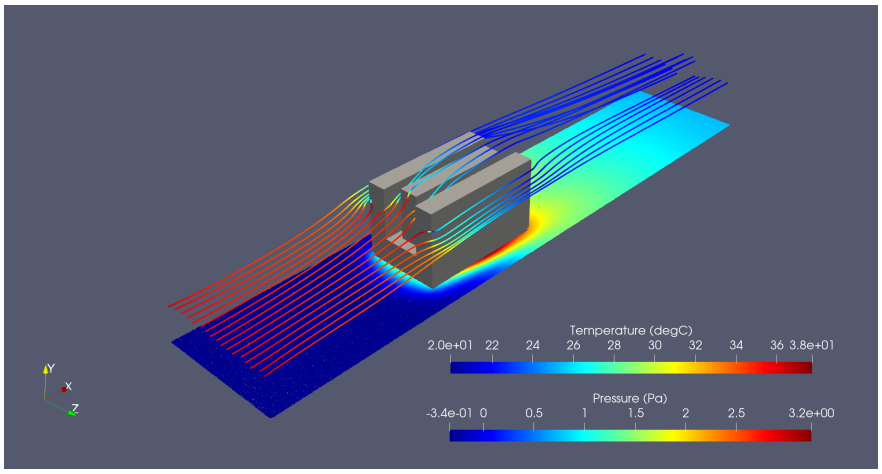


Figure 75: Streamlines colored with pressure and temperature profile in the fluid for optimal three fin geometry

Here, the 3-Fin heatsink was solved for arbitrary heat properties chosen such that the coupled conjugate heat transfer solution was possible. However, such approach causes issues when the conductivities are orders of magnitude different at the interface. We will revisit the conjugate heat transfer problem in tutorial 18 where we will solve the NVSwitch heatsink with real heat properties and also show how to use gPC for constructing surrogates for heatsink design optimization.

17 Case Study: FPGA Heat Sink with Laminar Flow for Single Geometry: Comparisons of Network Architectures, Optimizers and other schemes in Modulus

17.1 Introduction

In this example we show how some of the features in Modulus apply for a complicated FPGA heat sink design and solve the conjugate heat transfer. In this tutorial you would learn the following:

1. How to use Fourier Networks for complicated geometries with sharp gradients
2. How to solve problem with symmetry using symmetry boundary conditions
3. How to formulate velocity field as a vector potential (Exact continuity feature)
4. How different features and architectures in Modulus perform on a problem with complicated geometry

Prerequisites

This tutorial is very similar to the conjugate heat transfer problem that we have seen in the tutorials 13 and 16. We strongly recommend you to visit these tutorials (especially tutorial 13) for the details on the geometry generation, boundary conditions and monitor and validation domains. This tutorial would skip the description for the above stated processes and would instead focus more on the implementation of different features and the case study.

17.2 Problem Description

The geometry of the FPGA heatsink can be seen in figure 76. This particular geometry is challenging to simulate due to the thin closely spaced fins that causes sharp gradients which are particularly difficult to learn using regular fully connected neural network (slow convergence).

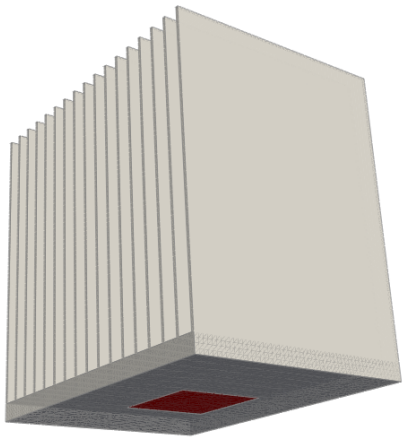


Figure 76: The FPGA heat sink geometry

We will be solving the conjugate heat transfer problem for the above geometry at Re=50. The dimensions of the geometry, as modeled in Modulus, are summarized in the Table 9:

Table 9: FPGA Dimensions

Dimension	Value
Heat Sink Base (l x b x h)	0.65 x 0.875 x 0.05
Fin dimension (l x b x h)	0.65 x 0.0075 x 0.8625
Heat Source (l x b)	0.25 x 0.25
Channel (l x b x h)	5.0 x 1.125 x 1.0

All the dimensions are scaled such that the channel height is 1 m. The temperature is scaled according to $\theta = T/273.15 - 1.0$. The channel walls are treated as adiabatic and the interface boundary conditions are applied at the fluid-solid interface. Other flow and thermal parameters are described in the Table 10

Table 10: Fluid and Solid Properties

Property	Fluid	Solid
Inlet Velocity (m/s)	1.0	NA
Density (kg/m^3)	1.0	1.0
Kinematic Viscosity (m^2/s)	0.02	NA
Thermal Diffusivity (m^2/s)	0.02	0.1
Thermal Conductivity $(W/m.K)$	1.0	1.0
Inlet Temperature (K)	273.15	NA
Heat Source Temperature Gradient (K/m)	409.725	NA

Case Setup

As suggested earlier, the case setup for this problem is very similar to the problem described in tutorial 13. Like tutorial 13 we will have 3 separate scripts for this problem for the domain definition, flow solver and heat solver. We will skip over the definitions of these files and would discuss only the major features introduced in this chapter.

Note: All the relevant domain, flow and heat solver files for this problem using various versions of features can be found at `examples/fpga/`.

17.3 Solver using Fourier Network Architecture

As described in the Theory section in tutorial 1, in Modulus, we can overcome the spectral bias of the neural networks by using the Fourier Networks. These networks have shown a significant improvement in results over the regular fully connected neural networks due to their ability to capture sharp gradients.

We do not need to make any special changes to the way the geometry and train domain are created while making changes to the neural network architectures. This also means the architecture is independent of the physics or parameterization being solved and can be applied to any other class of problems covered in the User Guide.

A note on frequencies: One main parameters of this networks are how we choose the frequencies. In Modulus we can choose from the spectrum we want to sample (full/partial/axis) and the number of frequencies in the spectrum. The optimal number of frequencies is an active field of research as one often needs to balance the accuracy benefits and the computational expense added due to use of extra Fourier features. For FPGA problem, we find that choosing 25 frequencies for laminar flow and 35 for turbulent flow gives a good balance between the two.

Moreover, for large domains where the length scale of the object of interest is significantly smaller than the max length in the domain (eg. bluff body simulations), we have found that normalizing the frequencies by that max length initializes the network better and speeds up convergence. Such modification can be achieved by setting the frequencies as `i/max_length` instead of just `i`. For the FPGA problem here, the `max_length` in the domain is the channel length in x-direction i.e. 5.0.

Below we show the solver file for the parameterized FPGA flow field simulation (`max_length = 5.0`), using the Fourier network architecture used with partial spectrum (axis and diagonal), 25 frequencies and with frequency normalization.

```
# import domain
from fpga_domain_parameterized import FPGAFlowTrain, FPGAFlowMonitor, FPGAFlowVal, FPGAFlowInference, nu, rho

# import Modulus library
from modulus.solver import Solver
from modulus.PDES.navier_stokes import IntegralContinuity, NavierStokes
from modulus.controller import ModulusController
from modulus.architecture.fourier_net import FourierNetArch

class FPGAFlowSolver(Solver):
  train_domain = FPGAFlowTrain
  monitor_domain = FPGAFlowMonitor
  val_domain = FPGAFlowVal
  inference_domain = FPGAFlowInference
  arch = FourierNetArch
```

```
  def __init__(self, **config):
    super(FPGAFlowSolver, self).__init__(**config)
    self.frequencies = ('axis,diagonal', [i/5. for i in range(25)])
    self.frequencies_params = ('axis,diagonal', [i/5. for i in range(20)])

    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z', 'HS_height', 'HS_length'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_fpga_fluid_flow_parameterized_fourier_net',
        'rec_results_cpu': True,
        'max_steps': 750000,
        'decay_steps': 7500
        })

if __name__ == '__main__':
  ctr = ModulusController(FPGAFlowSolver)
  ctr.run()
```

Listing 133: Using Fourier Networks for the FPGA problem

17.4 Leveraging Symmetry of the Problem

Whenever we have a symmetric geometry and we expect the variable fields to be symmetric, we can use symmetry boundary conditions about the plane or axis symmetry to minimize the computational expense of modeling the entire geometry. For the FPGA heat sink, we have such a plane of symmetry in the z-plane (Figure 77). The symmetry boundary conditions can be referred in Section 1.5.8 of the tutorial 1. Simulating the FPGA problem using symmetry, we achieve about 33% reduction in training time, compared to a training on the full domain.

For the FPGA problem where the plane of symmetry is z-plane, the boundary conditions stated in Section 1.5.8 can be translated to the following:

1. Variables which are odd functions w.r.t. `z` coordinate axis: `'w'`. Hence on symmetry plane `'w'=0`.
2. Variables which are even functions w.r.t. `z` coordinate axis: `'u'`, `'v'` components of velocity vector and scalar quantities like `'p'`, `'theta_s'` , `'theta_f'`. Hence on symmetry plane we set their normal derivative to 0. Eg. `'u__z'=0`.

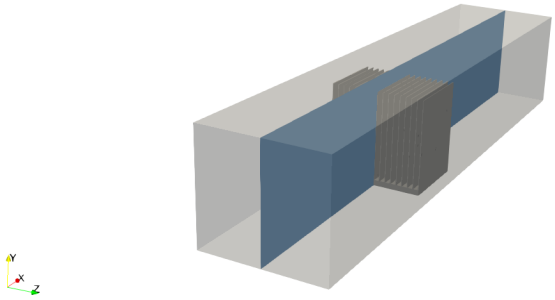


Figure 77: The FPGA heat sink with plane of symmetry

Below we show only the symmetry boundary conditions in the flow and heat training domains as rest other portion of train domain remain the same. (Full domain file can be accessed at `examples/fpga/`)

```
class FPGAFlowTrain(TrainDomain):
  def __init__(self, **config):
    super(FPGAFlowTrain, self).__init__()
    ...
    # symmetry BC
```

```
    symmetry = geo.boundary_bc(outvar_sympy={'w': 0, 'u__z': 0, 'v__z': 0, 'p__z': 0},
                               batch_size_per_area=500,
                               batch_per_epoch=5000,
                               criteria = Eq(z, channel_origin[2]+channel_dim[2]/2.))
    self.add(symmetry, name="SymmetricChannel")
    ...

class FPGAHeatTrain(TrainDomain):
  def __init__(self, **config):
    super(FPGAHeatTrain, self).__init__()
    ...
    # symmetry BC
    symmetrySolid = fpga.boundary_bc(outvar_sympy={'theta_s__z': 0},
             batch_size_per_area=500,
             criteria = Eq(z, channel_origin[2]+channel_dim[2]/2.),
             batch_per_epoch=5000)
    self.add(symmetrySolid, name="SymmetricChannelSolid")

    symmetryFluid = geo.boundary_bc(outvar_sympy={'theta_f__z': 0},
                                    batch_size_per_area=500,
                                    criteria = Eq(z, channel_origin[2]+channel_dim[2]/2.),
            batch_per_epoch=5000)
    self.add(symmetryFluid, name="SymmetricChannelFluid")
    ...
```

Listing 134: Using Symmetry boundary conditions for the FPGA problem

17.5 Imposing Exact Continuity

As described in the Theory section in tutorial 1, in Modulus, we can define the velocity field as a vector potential such that it is divergence free and satisfies continuity automatically. We can use this formulation for any class of flow problems covered in this User Guide regardless of the network architecture. However, we have found it to be most effective when using fully connected networks.

Below code shows the flow solver file for using exact continuity feature. No specific changes need to made to the train domain definition to use this feature.

```
# import domain
from fpga_domain import FPGAFlowTrain, FPGAFlowMonitor, FPGAFlowVal, nu, rho

from modulus.solver import Solver
from modulus.PDES.navier_stokes import NavierStokes, IntegralContinuity, Curl
from modulus.controller import ModulusController

# Define neural network
class FPGAFlowSolver(Solver):
  train_domain     = FPGAFlowTrain
  monitor_domain   = FPGAFlowMonitor
  val_domain       = FPGAFlowVal

  def __init__(self, **config):
    super(FPGAFlowSolver, self).__init__(**config)
    c = Curl(('a','b','c'), ('u','v','w'))
    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node()
                      + c.make_node())
    flow_net  = self.arch.make_node(name='flow_net',
                                    inputs=['x','y','z'],
                                    outputs=['a','b','c','p'])
    self.nets = [flow_net]

  @classmethod # Explain This
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_fpga_fluid_flow_exact_continuity',
        'rec_results': True,
        'rec_results_freq': 1000,
        })

if __name__ == '__main__':
  ctr = ModulusController(FPGAFlowSolver)
  ctr.run()
```

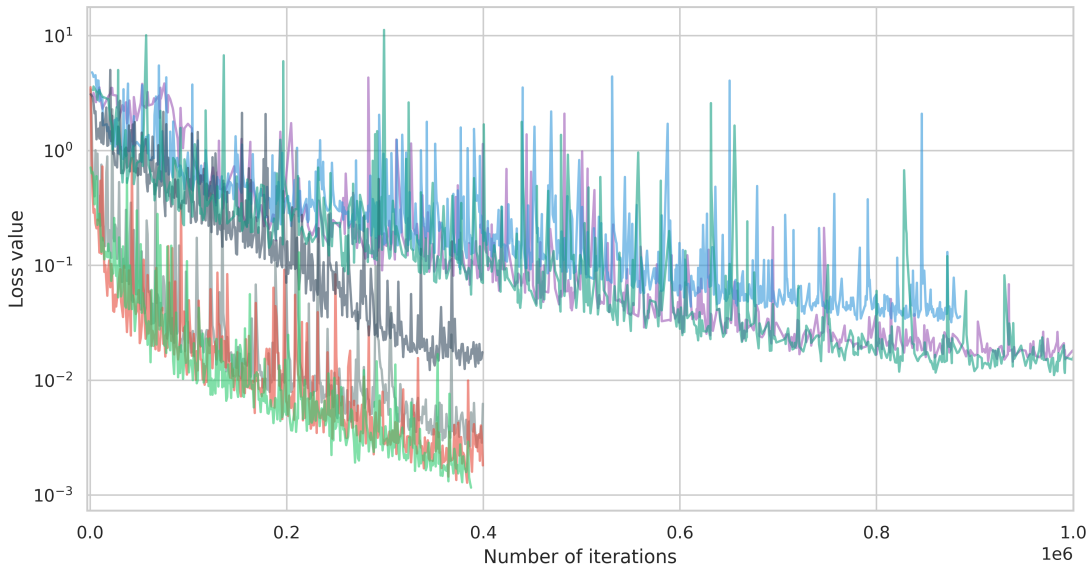
Listing 135: Defining velocity field as a vector potential

17.6 Results, Comparisons, and Summary

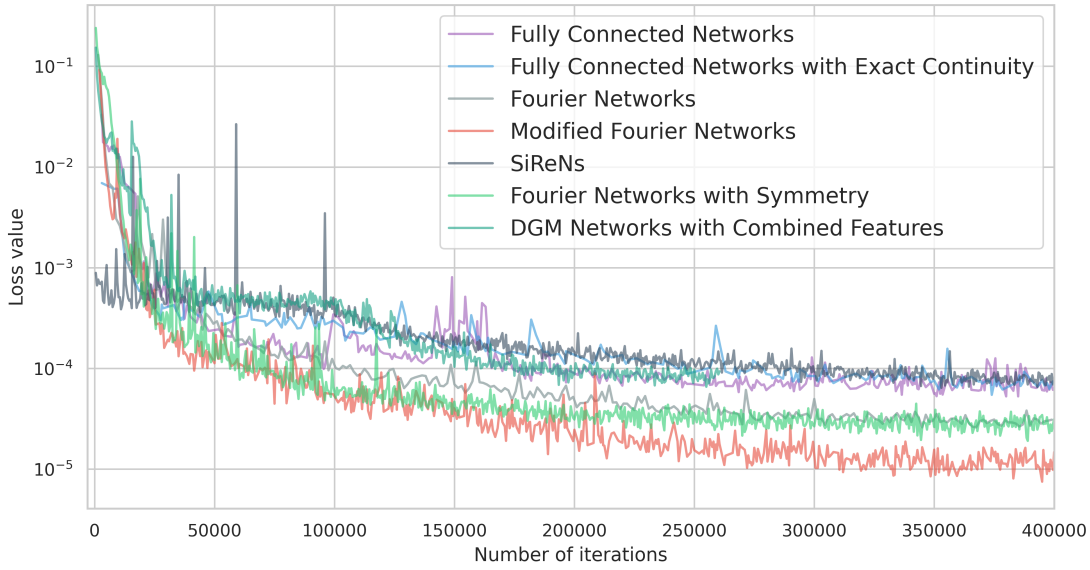
We have discussed several features in this chapter which might be a bit overwhelming. In the Table 11, we summarize these features and their applications and in Table 12, we summarize the important results of these features on this FPGA problem. Also, in Figure 78, we provide a comparison for the loss values from different runs.

Table 11: Summary of features introduced in this tutorial

Feature	Applicability to other problems	Comments
Fourier Networks	Applicable to all class of problems	Shown to be highly effective for problems involving sharp gradients. Modified Fourier network found to improve the performance one step further.
Symmetry	Applicable to all problems with a plane/axis of symmetry	Reduces the computational domain to half leading to significant speedup (33% reduction in training time compared to full domain)
Exact Continuity	Applicable to incompressible flow problems requiring solution to Navier Stokes equation	Gives better satisfaction of the continuity equation than Velocity-pressure formulation. Found to work best with standard fully connected networks. Also improves the accuracy of results in Fourier networks.
SiReNs	Applicable to all class of problems	Shown to be effective for problems with sharp gradients. However, we find that it does not outperform the Fourier Networks in terms of accuracy.
DGM Networks with Global LR annealing, Global Adaptive Activations and Halton Sequences	Applicable to all class of problems	Improves accuracy compared to regular fully connected networks.



(a) flow



(b) heat

Figure 78: A comparison for the loss values from different runs as listed in Table 12

Table 12: Comparison of pressure drop and peak temperatures from various runs

Case Description	$P_{drop}\ (Pa)$	$T_{peak}\ (^{\circ}C)$
Modulus: Fully Connected Networks	29.24	77.90
Modulus: Fully Connected Networks with Exact Continuity	28.92	90.63
Modulus: Fourier Networks	29.19	77.08
Modulus: Modified Fourier Networks	29.23	80.96
Modulus: SiReNs	29.21	76.54
Modulus: Fourier Networks with Symmetry	29.14	78.56
Modulus: DGM Networks with Global LR annealing, Global Adaptive Activations, and Halton Sequences	29.10	76.86
OpenFOAM Solver	28.03	76.67
Commercial Solver	28.38	84.93

18 Industrial Heat Sink simulations

18.1 Introduction

In this tutorial, we will use Modulus to conduct a thermal simulation of NVIDIA's NVSwitch heatsink. You will learn the following:

1. How to use hFTB algorithm to solve conjugate heat transfer problems
2. How to make a script to evaluate results on mesh grids
3. How to build a gPC-Based Surrogate via Transfer Learning

Prerequisites

This tutorial assumes you have completed tutorial 6 on transient Navier-Stokes via moving time window as well as the tutorial 13 on conjugate heat transfer.

18.2 Problem Description

Here, we solve the conjugate heat transfer problem of NVIDIA's NVSwitch heat sink as shown in 79. Similar to the previous FPGA problem, the heat sink is placed in a channel with inlet velocity similar to its operating conditions. This case differs from our previous conjugate heat transfer problems because we will be using the real heat properties for atmospheric air and copper as the heat sink material.

Using real heat properties causes an issue on the interface between the solid and fluid because the conductivity is around 4 orders of magnitude different (Air: 0.0261 $W/m.K$ and Copper: 385 $W/m.K$). If attempting to solve in a fully coupled manner as before this will cause issues with convergence. To remedy this, we have implemented a static conjugate heat transfer approached referred to as heat transfer coefficient forward temperature backward or hFTB [34]. This method works by iteratively solving for the heat transfer in the fluid and solid where they are one-way coupled. Using the hFTB method, we assign Robin boundary conditions on the solid interface and Dirichlet boundaries for the fluid. The simulation starts by giving an initial guess for the solid temperature and uses a hyper parameter h for the Robin boundary conditions. We give a description of the algorithm here 1 and refer the readers to more complete description here [34].

Algorithm 1: hFTB algorithm

Solve fluid with interface Dirichlet=INITIAL_TEMP;
for *i :=1 to NUMBER_CYCLES* **do**
    AMBIENT_TEMP = FLUID_TEMP - FLUID_FLUX / h;
    Solve Solid with interface Robin=SOLID_FLUX - h(SOLID_TEMP - AMBIENT_TEMP);
    Solve Fluid with Dirichlet=SOLID_TEMP;
end

18.3 Case Setup

The case setup for this problem is similar to the FPGA and three fin examples (covered in tutorials 13 and 17) however we will construct multiple train domains to implement the hFTB method. We will also show a script that can be used to evaluate the results on a mesh grid.

Note: The python script for this problem can be found at `examples/limerock/`.

18.3.1 Defining Domain

In this case setup we will skip over several sections of the code and only focus on the portions related to the hFTB algorithm. We will assume that the reader is familiar with how to set up the flow simulation from previous tutorials. We also will not go into the details about the geometry construction and assume we can get all relevant information from the module `examples/limerock/limerock_hFTB/stl_to_geo.py`. We will begin the code description by defining the parameters of the simulation and importing all needed modules.

```
# import geometry
from stl_to_geo import LimeRock
```

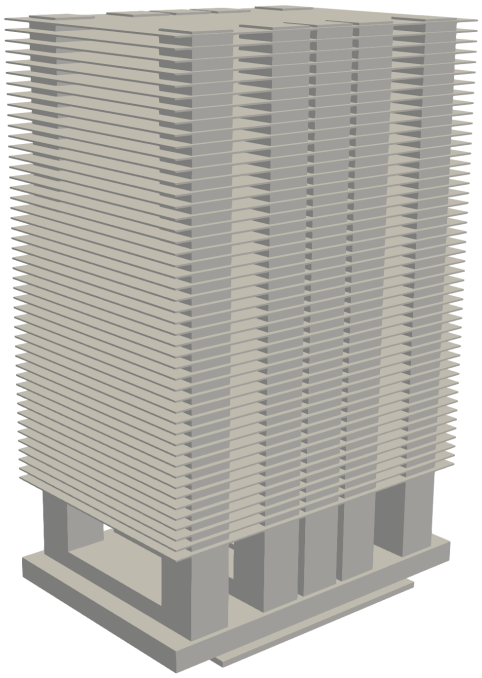


Figure 79: NVSwitch heat sink geometry

```
# import Modulus library
from sympy import Symbol, Eq, tanh, Abs, Or
import numpy as np
import tensorflow as tf
from modulus.dataset import TrainDomain, ValidationDomain, MonitorDomain, InferenceDomain
from modulus.data import Validation, Monitor, Inference
from modulus.sympy_utils.geometry_3d import Box, Channel, Plane
from modulus.csv_utils.csv_rw import csv_to_dict

# make limerock
limerock = LimeRock()

##############
# Real Params
##############
# fluid params
fluid_viscosity = 1.84e-05 # kg/m-s
fluid_density = 1.1614 # kg/m3
fluid_specific_heat = 1005 # J/(kg K)
fluid_conductivity = 0.0261 # W/(m K)

# copper params
copper_density = 8930 # kg/m3
copper_specific_heat = 385 # J/(kg K)
copper_conductivity = 385 # W/(m K)

# boundary params
inlet_velocity = 5.7 # m/s
inlet_temp = 0 # K

# source
source_term = 2127.71 # K/m
source_origin = (-0.061667, -0.15833, limerock.geo_bounds_lower[2])
source_dim = (0.1285, 0.31667, 0)

#################
# Non dim params
#################
length_scale = 0.0575 # m
velocity_scale = 5.7 # m/s
time_scale = length_scale/velocity_scale # s
density_scale = 1.1614 # kg/m3
mass_scale = density_scale*length_scale**3 # kg
pressure_scale = mass_scale / (length_scale * time_scale**2) # kg / (m s**2)
```

```
temp_scale = 273.15 # K
watt_scale = (mass_scale * length_scale**2) / (time_scale**3) # kg m**2 / s**3
joule_scale = (mass_scale * length_scale**2) / (time_scale**2) # kg * m**2 / s**2

############################
# Nondimensionalization Params
############################
# fluid params
nd_fluid_viscosity      = fluid_viscosity / (length_scale**2 / time_scale) # need to divide by density to get
    previous viscosity
nd_fluid_density        = fluid_density / density_scale
nd_fluid_specific_heat  = fluid_specific_heat / (joule_scale / (mass_scale * temp_scale))
nd_fluid_conductivity   = fluid_conductivity / (watt_scale / (length_scale * temp_scale))
nd_fluid_diffusivity    = nd_fluid_conductivity / (nd_fluid_specific_heat * nd_fluid_density)

# copper params
nd_copper_density       = copper_density / (mass_scale / length_scale**3)
nd_copper_specific_heat = copper_specific_heat / (joule_scale / (mass_scale * temp_scale))
nd_copper_conductivity  = copper_conductivity / (watt_scale / (length_scale * temp_scale))
nd_copper_diffusivity   = nd_copper_conductivity / (nd_copper_specific_heat * nd_copper_density)

# boundary params
nd_inlet_velocity  = inlet_velocity/velocity_scale
nd_volumetric_flow = limerock.inlet_area * nd_inlet_velocity
nd_inlet_temp      = inlet_temp / temp_scale
nd_source_term     = source_term / (temp_scale / length_scale)
```

Listing 136: Defining simulation parameters

Note: We non-dimensionalize all parameters so that the scales for velocity, temperature, and pressure are roughly in the range 0-1. Such non dimensionalization trains the Neural network more efficiently.

Now we will set up a train domain to only solve for the temperature in the fluid given a Dirichlet boundary condition on the solid. This will be the first stage of the hFTB method.

```
class Cycle1hFTBTrain(TrainDomain):
  name = 'cycle_1'
  nr_iterations = 1

  def __init__(self, **config):
    super(Cycle1hFTBTrain, self).__init__()
    scale_batch = 4

    # get initial temp for hFTB algorithm
    initial_theta_f = config['config'].initial_t # This can be any temp but 1.0 is a good guess for this
     problem

    # inlet
    inletBC = limerock.inlet.boundary_bc(outvar_sympy={'theta_f': nd_inlet_temp},
                                         batch_size_per_area=50//scale_batch,
                                         lambda_sympy={'lambda_theta_f': 1000.0},
                                         batch_per_epoch=2000*scale_batch,
                                         criteria=Eq(x, limerock.geo_bounds_lower[0]))
    self.add(inletBC, name="Inlet_Cycle1")
    print("made inlet")

    # outlet
    outletBC = limerock.outlet.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                           batch_size_per_area=50//scale_batch,
                                           lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                           batch_per_epoch=2000*scale_batch,
                                           criteria=Eq(x, limerock.geo_bounds_upper[0]))
    self.add(outletBC, name="Outlet_Cycle1")
    print("made outlet")

    # channel walls insulating
    walls = limerock.geo.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                     batch_size_per_area=50//scale_batch,
                                     lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                     criteria=Eq(y, limerock.geo_bounds_lower[1])|Eq(z, limerock.
     geo_bounds_lower[2])|Eq(y, limerock.geo_bounds_upper[1])|Eq(z, limerock.geo_bounds_upper[2]),
                                     batch_per_epoch=2000*scale_batch)
    self.add(walls, name="ChannelWalls_Cycle1")
    print("made channel walls")

    # flow interior low res away from heat sink
    interiorF_lr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                            bounds=limerock.geo_bounds,
                                            batch_size_per_area=120//scale_batch,
```

```
                                        lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                        criteria=(x < limerock.heat_sink_bounds[0]) | (x > limerock.
    heat_sink_bounds[1]),
                                        batch_per_epoch=2000*scale_batch)
    self.add(interiorF_lr, name="FlowInterior_LR_Cycle1")
    print("made lr flow interior")

    # flow interiror high res near heat sink
    interiorF_hr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                        bounds=limerock.geo_hr_bounds,
                                        batch_size_per_area=750//scale_batch,
                                        lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                        batch_per_epoch=2000*scale_batch)
    self.add(interiorF_hr, name="FlowInterior_HR_Cycle1")
    print("made hr flow interior")

    # fluid solid interface
    interface = limerock.geo_solid.boundary_bc(outvar_sympy={'theta_f': initial_theta_f},
                                            batch_size_per_area=50//scale_batch,
                                            lambda_sympy={'lambda_theta_f': 100.0},
                                            criteria=z>limerock.geo_bounds_lower[2],
                                            batch_per_epoch=2000*scale_batch)
    self.add(interface, name="Interface_Cycle1")
    print("made solid flow interface")
```

Listing 137: Defining Train Domain

After getting this initial solution for the temperature in the fluid we solve for the main loop of the hFTB algorithm. Now we will solve for both the fluid and solid in a one way coupled manner. The Robin boundary conditions for the solid are coming from the previous iteration of the fluid solution.

```
class CycleNhFTBTrain(TrainDomain):
  name = 'cycle_n'
  nr_iterations = 10

  def __init__(self, **config):
    super(CycleNhFTBTrain, self).__init__()
    scale_batch = 4

    # inlet
    inletBC = limerock.inlet.boundary_bc(outvar_sympy={'theta_f': nd_inlet_temp},
                                        batch_size_per_area=50//scale_batch,
                                        lambda_sympy={'lambda_theta_f': 1000.0},
                                        batch_per_epoch=2000*scale_batch,
                                        criteria=Eq(x, limerock.geo_bounds_lower[0]))
    self.add(inletBC, name="Inlet")
    print("made inlet")

    # outlet
    outletBC = limerock.outlet.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                        batch_size_per_area=50//scale_batch,
                                        lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                        batch_per_epoch=2000*scale_batch,
                                        criteria=Eq(x, limerock.geo_bounds_upper[0]))
    self.add(outletBC, name="Outlet")
    print("made outlet")

    # channel walls insulating
    walls = limerock.geo.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                    batch_size_per_area=50//scale_batch,
                                    lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                    criteria=Eq(y, limerock.geo_bounds_lower[1])|Eq(z, limerock.
    geo_bounds_lower[2])|Eq(y, limerock.geo_bounds_upper[1])|Eq(z, limerock.geo_bounds_upper[2]),
                                    batch_per_epoch=2000*scale_batch)
    self.add(walls, name="ChannelWalls")
    print("made channel walls")

    # flow interior low res away from heat sink
    interiorF_lr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                        bounds=limerock.geo_bounds,
                                        batch_size_per_area=120//scale_batch,
                                        lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                        criteria=(x < limerock.heat_sink_bounds[0]) | (x > limerock.
    heat_sink_bounds[1]),
                                        batch_per_epoch=2000*scale_batch)
    self.add(interiorF_lr, name="FlowInterior_LR")
    print("made lr flow interior")

    # flow interiror high res near heat sink
```

```
    interiorF_hr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                            bounds=limerock.geo_hr_bounds,
                                            batch_size_per_area=750//scale_batch,
                                            lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                            batch_per_epoch=2000*scale_batch)
    self.add(interiorF_hr, name="FlowInterior_HR")
    print("made hr flow interior")

    # diffusion dictionaries
    diffusion_outvar_sympy = {'diffusion_theta_s': 0,
                              'compatibility_theta_s_x_y': 0,
                              'compatibility_theta_s_x_z': 0,
                              'compatibility_theta_s_y_z': 0,
                              'integrate_diffusion_theta_s_x': 0,
                              'integrate_diffusion_theta_s_y': 0,
                              'integrate_diffusion_theta_s_z': 0}
    diffusion_lambda_sympy = {'lambda_diffusion_theta_s': 1000000.0,
                              'lambda_compatibility_theta_s_x_y': 1.0,
                              'lambda_compatibility_theta_s_x_z': 1.0,
                              'lambda_compatibility_theta_s_y_z': 1.0,
                              'lambda_integrate_diffusion_theta_s_x': 1.0,
                              'lambda_integrate_diffusion_theta_s_y': 1.0,
                              'lambda_integrate_diffusion_theta_s_z': 1.0}

    # solid interior
    interiorS = limerock.geo_solid.interior_bc(outvar_sympy=diffusion_outvar_sympy,
                                               bounds=limerock.geo_hr_bounds,
                                               batch_size_per_area=5000//scale_batch,
                                               lambda_sympy=diffusion_lambda_sympy,
                                               batch_per_epoch=2000*scale_batch)
    self.add(interiorS, name="SolidInterior")
    print("made solid interior")

    # limerock base
    sharpen_tanh = 60.0
    source_func_xl = (tanh(sharpen_tanh*(x - source_origin[0])) + 1.0)/2.0
    source_func_xh = (tanh(sharpen_tanh*((source_origin[0]+source_dim[0]) - x)) + 1.0)/2.0
    source_func_yl = (tanh(sharpen_tanh*(y - source_origin[1])) + 1.0)/2.0
    source_func_yh = (tanh(sharpen_tanh*((source_origin[1]+source_dim[1]) - y)) + 1.0)/2.0
    gradient_normal = nd_source_term * source_func_xl * source_func_xh * source_func_yl * source_func_yh
    limerock_base = limerock.geo_solid.boundary_bc(outvar_sympy={'normal_gradient_flux_theta_s':
     gradient_normal},
                                                   lambda_sympy={'lambda_normal_gradient_flux_theta_s': 10.0},
                                                   batch_size_per_area=500//scale_batch, # NOTE I put this a
     guess for the batch size and am still looking at it
                                                   criteria=Eq(z,limerock.geo_bounds_lower[2]),
                                                   batch_per_epoch=2000*scale_batch)
    self.add(limerock_base, name="HeatSinkBase")
    print("made base source")

    # fluid solid interface
    interface = limerock.geo_solid.boundary_bc(outvar_sympy={'dirichlet_theta_s_theta_f': 0,
                                                             'robin_theta_s': 0},
                                               batch_size_per_area=50//scale_batch,
                                               lambda_sympy={'lambda_dirichlet_theta_s_theta_f': 100.0,
                                                             'lambda_robin_theta_s': 1.0},
                                               criteria=z>limerock.geo_bounds_lower[2],
                                               batch_per_epoch=2000*scale_batch)
    self.add(interface, name="Interface")
    print("made solid flow interface")
```

Listing 138: Defining Train Domain

18.3.2 Sequence Solver

Now we setup the solver. Similar to the moving time window implementation (tutorial 6), we construct a separate neural network that stores the thermal solution from the previous cycles fluid solution. We also define a `custom_update_op` that will restart the learning rate decay and update networks weights after each cycle. We also add an option to allow for gradient aggregation with the default set to 2. This can be used to increase the batch size. We suggest that this problem is either run on 4 GPUs or this gradient aggregation is set to 8. Details on running with multi-GPUs and multi-nodes can be found in tutorial 19 and the details on using gradient aggregation can be found in tutorial 12.

```
# import domain
from limerock_domain import Cycle1hFTBTrain, CycleNhFTBTrain, LRHeatMonitor, LRHeatInference, limerock,
    nd_fluid_viscosity, nd_fluid_density, nd_fluid_diffusivity, nd_copper_conductivity, nd_fluid_conductivity,
    nd_copper_diffusivity
```

```
# import vector flux diffusion equations
from flux_diffusion import FluxDiffusion, FluxGradNormal, FluxIntegrateDiffusion, FluxRobin, Dirichlet

# import Modulus library
from modulus.solver import Solver
from modulus.PDES.navier_stokes import GradNormal
from modulus.PDES.advection_diffusion import AdvectionDiffusion
from modulus.controller import ModulusController
from modulus.architecture.fourier_net import FourierNetArch
from modulus.node import Node
from modulus.variables import Key, Variables
from modulus.config import str2bool
from modulus.optimizer import AdamOptimizerGradAgg
grad_agg_freq = 2  # number of gradient aggregations

class LRHeatSolver(Solver):
  seq_train_domain = [Cycle1hFTBTrain, CycleNhFTBTrain]
  monitor_domain = LRHeatMonitor
  arch = FourierNetArch
  optimizer = AdamOptimizerGradAgg

  def __init__(self, **config):
    super(LRHeatSolver, self).__init__(**config)

    self.arch.frequencies = ('axis', [i/5. for i in range(50)])

    # configs for the hFTB algorithm
    h = config['config'].h # This can be arbitrarily positive number
    initial_theta_f = config['config'].initial_t # This can be any temp that is in the range possible solutions,
      say between 0-1.

    # make list of equations
    self.equations = (AdvectionDiffusion(T='theta_f', rho=nd_fluid_density, D=nd_fluid_diffusivity, dim=3, time
     =False).make_node(stop_gradients=['u', 'v', 'w'])
                    + GradNormal('theta_f', dim=3, time=False).make_node()
                    + FluxDiffusion(D=nd_copper_diffusivity).make_node()
                    + FluxIntegrateDiffusion().make_node(stop_gradients=['flux_theta_s_x', 'flux_theta_s_y', '
     flux_theta_s_z'])
                    + FluxGradNormal().make_node()
                    + FluxRobin(theta_f_conductivity=nd_fluid_conductivity, theta_s_conductivity=
     nd_copper_conductivity, h=h).make_node(stop_gradients=['theta_f_prev_step','theta_f_prev_step__x','
     theta_f_prev_step__y','theta_f_prev_step__z'])
                    + Dirichlet(lhs='theta_f', rhs='theta_s').make_node(stop_gradients=['theta_s']))

    # make networks
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    solid_heat_net = self.arch.make_node(name='solid_heat_net',
                                         inputs=['x', 'y', 'z'],
                                         outputs=['theta_s'])
    flux_heat_net = self.arch.make_node(name='flux_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['flux_theta_s_x',
                                                 'flux_theta_s_y',
                                                 'flux_theta_s_z'])
    flow_heat_net = self.arch.make_node(name='flow_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['theta_f'])
    flow_heat_net_prev_step = self.arch.make_node(name='flow_heat_net_prev_step',
                                                  inputs=['x', 'y', 'z'],
                                                  outputs=['theta_f_prev_step'])
    self.nets = [flow_net, solid_heat_net, flux_heat_net, flow_heat_net, flow_heat_net_prev_step]

  def custom_update_op(self):
    # zero train step op
    global_step = [v for v in tf.get_collection(tf.GraphKeys.VARIABLES) if 'global_step' in v.name][0]
    zero_step_op = tf.assign(global_step, tf.zeros_like(global_step))

    # make update op that sets weights from_flow_net to flow_net_prev_step
    prev_assign_step = [zero_step_op]
    flow_heat_net_variables = [v for v in tf.trainable_variables() if 'flow_heat_net/' in v.name]
    flow_heat_net_prev_step_variables = [v for v in tf.trainable_variables() if 'flow_heat_net_prev_step' in v.
     name]
    for v, v_prev_step in zip(flow_heat_net_variables, flow_heat_net_prev_step_variables):
      prev_assign_step.append(tf.assign(v_prev_step, v))
    prev_assign_step = tf.group(*prev_assign_step)
    return prev_assign_step

  @classmethod
```

```
  def add_options(cls, group):
    group.add_argument('--initial_t',
                       help='initial tempurature',
                       type=float,
                       default=0.05)
    group.add_argument('--h',
                       help='H values',
                       type=float,
                       default=500.0)

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'initialize_network_dir': './network_checkpoint_limerock_fluid_flow_real',
        'network_dir': './network_checkpoint_limerock_heat_hFTB',
        'added_config_dir': 'initial_t,h',
        'rec_results_cpu': True,
        'max_steps': 150000*grad_agg_freq,
        'decay_steps': 1500*grad_agg_freq,
        'rec_results_freq': 1000,
        'start_lr': 1e-3,
        'grad_agg_freq': grad_agg_freq
        })

if __name__ == '__main__':
  ctr = ModulusController(LRHeatSolver)
  ctr.run()
```

Listing 139: Defining the Solver

18.3.3 Mesh Grid Evaluation Script

Evaluating with the Inference Domain only allows the results to be saved as a .vti point cloud. Sometimes it is beneficial to visualize the results on a mesh. The following scripts shows how this is possible using the `stream` method. The script starts by defining a solver class like before however we are now giving a specific network checkpoint to restore from (`./network_checkpoint_limerock_heat_hFTB_2/initial_t.0.05/h.500.0/cycle_n_0009`). Next we construct an function `thermal_stream` that will allow us to evaluate results on arbitrary points. The remainder of the script is devoted to making a mesh grid to evaluate on and saving the results. Note that we use the SDF to zero the temperature evaluations outside the solid.

```
# import domain
from stl_to_geo import LimeRock

# import needed packages
import sys
from sympy import Symbol
import numpy as np
from evtk.hl import imageToVTK

# import Modulus
from modulus.solver import Solver
from modulus.sympy_utils.geometry_3d import Box, Sphere
from modulus.dataset import TrainDomain
from modulus.controller import ModulusController
from modulus.node import Node
from modulus.variables import Variables, Key
from modulus.config import str2bool
from modulus.sympy_utils.numpy_printer import np_lambdify
from modulus.architecture.fourier_net import FourierNetArch

# make limerock
limerock = LimeRock()

# construct solver class for loading neural network simulation
class LRSolver(Solver):
  train_domain = TrainDomain # blank domain
  arch = FourierNetArch

  def __init__(self, **config):
    super(LRSolver, self).__init__(**config)

    self.arch.frequencies = ('axis', [i/5. for i in range(50)])

    # make networks
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
```

```
                                    outputs=['u', 'v', 'w', 'p'])
    solid_heat_net = self.arch.make_node(name='solid_heat_net',
                                         inputs=['x', 'y', 'z'],
                                         outputs=['theta_s'])
    flux_heat_net = self.arch.make_node(name='flux_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['flux_theta_s_x',
                                                 'flux_theta_s_y',
                                                 'flux_theta_s_z'])
    flow_heat_net = self.arch.make_node(name='flow_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['theta_f'])
    flow_heat_net_prev_step = self.arch.make_node(name='flow_heat_net_prev_step',
                                                  inputs=['x', 'y', 'z'],
                                                  outputs=['theta_f_prev_step'])
    self.nets = [flow_net, solid_heat_net, flux_heat_net, flow_heat_net, flow_heat_net_prev_step]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_limerock_heat_hFTB_2/initial_t.0.05/h.500.0/cycle_n_0009',
        'rec_results_cpu': True,
        })

# make controler to start streaming results
ctr = ModulusController(LRSolver)

# function for evaluating flow
thermal_stream = ctr.stream(['x','y','z'], ['theta_s', 'theta_f', 'u', 'v', 'w', 'p'])

# function for evaluating SDF
input_params = {'x': None,
                'y': None,
                'z': None}
limerock_sdf_fn = np_lambdify(limerock.geo_solid.sdf, input_params)

# make plot data
nx, ny, nz = 128, 128, 512
ncells = nx * ny * nz
origin = [x[0] for x in limerock.geo_hr_bounds.values()]
spacing = [(x[1] - x[0])/y for x, y in zip(limerock.geo_hr_bounds.values(), [nx, ny, nz])]
mesh_x, mesh_y, mesh_z = np.meshgrid(np.linspace(origin[0], nx*spacing[0]+origin[0], nx),
                                     np.linspace(origin[1], ny*spacing[1]+origin[1], ny),
                                     np.linspace(origin[2], nz*spacing[2]+origin[2], nz),
                                     indexing='ij')
mesh_x = np.expand_dims(mesh_x, axis=-1)
mesh_y = np.expand_dims(mesh_y, axis=-1)
mesh_z = np.expand_dims(mesh_z, axis=-1)

# evaluate SDF and flow on points
limerock_sdf = {'sdf': limerock_sdf_fn(x=mesh_x, y=mesh_y, z=mesh_z)}
limerock_thermal = thermal_stream({'x': mesh_x, 'y': mesh_y, 'z': mesh_z})
limerock_data = {**limerock_sdf, **limerock_thermal}
cell_data = {}
for key, value in limerock_data.items():
  if str(key) in ['theta_f', 'u', 'v', 'w', 'p']:
    cell_data[str(key)] = np.ascontiguousarray((value*np.heaviside(-limerock_data['sdf'], 0))[...,0])
  elif str(key) in ['theta_s']:
    cell_data[str(key)] = np.ascontiguousarray((value*np.heaviside(limerock_data['sdf'], 0))[...,0])
  else:
    cell_data[str(key)] = np.ascontiguousarray(value[...,0])

# save everything to vti file
imageToVTK("limerock_evaluation", origin, spacing, cellData=cell_data)
```

Listing 140: Evaluating results on a Mesh grid

18.4 Results and Post-processing

To confirm the accuracy of our model, we compare the Modulus results for pressure drop and peak temperature with the OpenFOAM and a commercial solver results, reported in Table 14. The results show good accuracy achieved by the hFTB method. Table 14 demonstrates the impact of mesh refinement on the solution of the commercial solver where with increasing mesh density and mesh quality, the commercial solver results show convergence towards the Modulus results. We show a visualization of the heat sink temperature profile in Figure 80.

Table 13: A comparison for the solver and Modulus results for NVSwitch pressure drop and peak temperature.

Property	OpenFOAM	Commercial Solver	Modulus
Pressure Drop (Pa)	133.96	137.50	150.25
Peak Temperature $(^{\circ}C)$	93.41	95.10	97.35

Table 14: Commercial solver mesh refinement results for NVSwitch pressure drop and peak temperature.

Number of elements	Commercial solver pressure drop (Pa)	Modulus pressure drop (Pa)	Absolute difference (%)	Commercial solver peak temperature $(^{\circ}C)$	Modulus peak temperature $(^{\circ}C)$	Absolute difference (%)
22.4 M	81.27	150.25	84.88	97.40	97.35	0.05
24.7 M	111.76	150.25	34.44	95.50	97.35	1.94
26.9 M	122.90	150.25	22.25	95.10	97.35	2.36
30.0 M	132.80	150.25	13.14	-	-	-
32.0 M	137.50	150.25	9.27	-	-	-

18.5 gPC-Based Surrogate Modeling Accelerated via Transfer Learning

Previously in Chapter 16 we showed that by parameterizing the input of our neural network, we can solve for multiple design parameters in a single run and use that parameterized network for design optimization. Here in this section, we introduce another approach for parameterization and design optimization, which is based on constructing a surrogate using the solution obtained from a limited number of non-parameterized neural network models. Compared to the parameterized network approach that is limited to the CSG module, this approach can be used for parameterization of both constructive solid and STL geometries, and additionally, can offer improved accuracy specially for cases with a high-dimensional parameter space and also in cases where some or all of the design parameters are discrete. However, this approach requires training of multiple neural networks and may require multi-node resources.

We specifically focus on surrogates based on the generalized Polynomial Chaos (gPC) expansions. The gPC is an efficient tool for uncertainty quantification using limited data, and in introduced in Section 1.8. We start off by generating the required number of realizations form the parameter space using a low-discrepancy sequence such as Halton or Sobol. Next, for each realization, we train a separate neural network model. Note that these trainings are independent from each other and therefore, this training step is embarrassingly parallel and can be done on multiple GPUs or nodes. Finally, we train a gPC surrogate that maps the parameter space to the quantities of interest (e.g., pressure drop and peak temperature in the heat sink design optimization problem).

In order to reduce the computational cost of this approach associated with training of multiple models, we use transfer learning, that is, once a model is fully trained for a single realization, it is used for initialization of the other models, and this can significantly reduce the total time to convergence. Transfer learning has been previously introduced in Chapter 14.

Here, to illustrate the gPC surrogate modeling accelerated via transfer learning, we consider the NVIDIA's NVSwitch heat sink introduced in the previous section, and limit the geometry parameters to four fin cut angle parameters, as shown in Figure 81. We then construct a pressure drop surrogate. Similarly, one can also construct a surrogate for the peak temperature and use these two surrogates for design optimization of this heat sink.

The scripts for this example are in the `limerock/limerock_4params_pce_surrogate` directory. Following Section 1.8, we generate 30 geometry realizations according to a Halton sequence by running `sample_generator.py`, as follows

```
# import libraries
import numpy as np
import chaospy

# define parameter ranges
fin_front_top_cut_angle_ranges = (0., np.pi/6.)
fin_front_bottom_cut_angle_ranges = (0., np.pi/6.)
fin_back_top_cut_angle_ranges = (0., np.pi/6.)
fin_back_bottom_cut_angle_ranges = (0., np.pi/6.)

# generate samples
samples = chaospy.generate_samples(order=30,
                                   domain=np.array([fin_front_top_cut_angle_ranges,
                                                    fin_front_bottom_cut_angle_ranges,
```

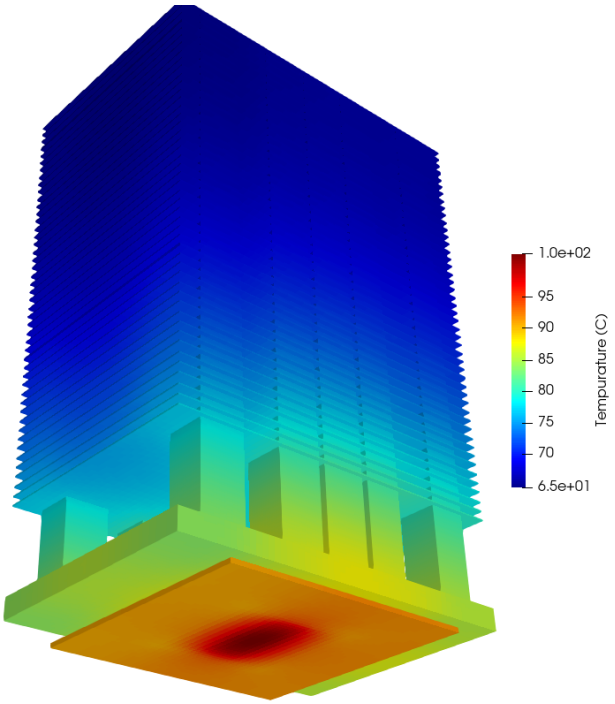


Figure 80: NVSwitch Solid Temperature

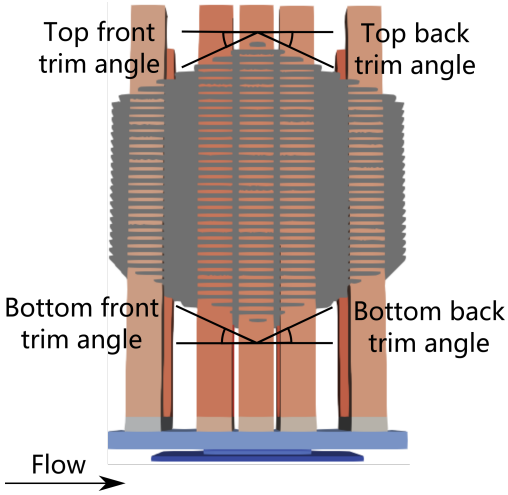


Figure 81: NVSwitch heat sink geomtry parameterization. Each parameter ranges between 0 and $\pi/6$.

```
                                                    fin_back_top_cut_angle_ranges,
                                                    fin_back_bottom_cut_angle_ranges]).T,
                                        rule="halton")
samples = samples.T
np.random.shuffle(samples)
np.savetxt('samples.txt', samples)
```

Listing 141: Generating geometry realizations using a Halton sequence

We then train a separate flow network for each of these realizations using transfer learning. To do this, we change the `sample_id` variable in `stl_to_geo.py`, and then run `limerock_flow_solver.py`. This is repeated until all of the geometry realizations are covered. These flow models are initialized using the trained network for the base geometry (as shown in Figure 79), and are trained for a fraction of the total training iterations for the base geometry, with a smaller learning rate and a faster learning rate decay. This is because we only need to fine-tune these models as opposed to training them from the scratch. Please note that, before we launch the transfer learning runs, a flow network for the base geometry needs to be fully trained.

```
@classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'initialize_network_dir': './network_checkpoint_limerock_fluid_flow_real',  # network dir from a full
     run
        'network_dir': './network_checkpoint_limerock_fluid_flow_real_TL_' + str(sample_id),
        'rec_results_cpu': True,
        'max_steps': 200000,
        'decay_steps': 2500,
        'start_lr': 2.5e-4,
        # 'max_steps': 1000000, # if full run
        # 'decay_steps': 10000, # if full run
        # 'start_lr': 5e-4,  # if full run
        })
```

Listing 142: Training configs for a transfer learning model

Figure 82 shows the front and back pressure results for different runs. It is evident that the pressure has converged faster in the transfer learning runs compared to the base geometry full run, and that transfer learning has reduced the total time to convergence by a factor of 5.

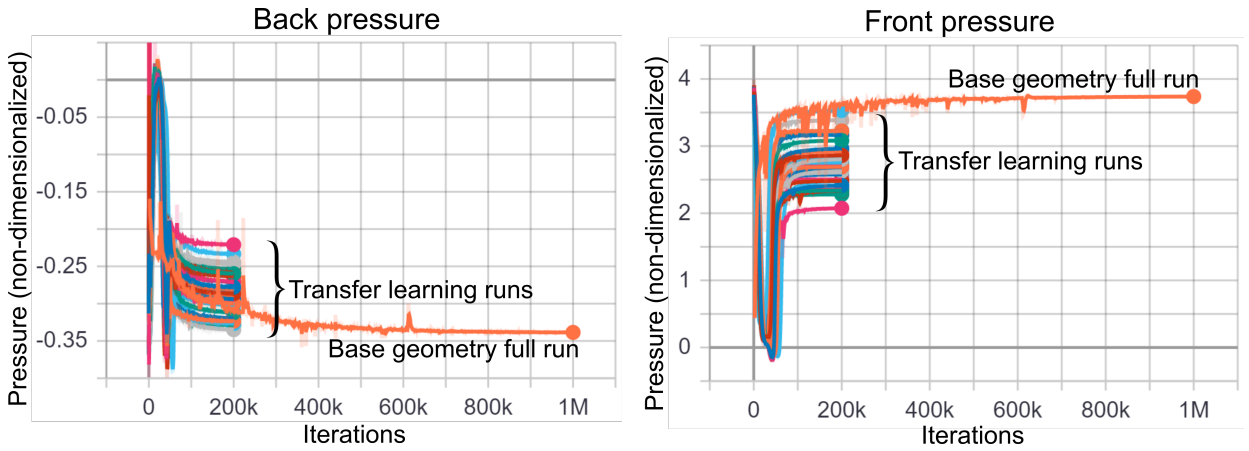


Figure 82: NVSwitch front and back pressure convergence results for different geometries using transfer learning.

Finally, we randomly divide the pressure drop data obtained from these models into training and test sets, and construct a gPC surrogate by running `limerock_pressure_drop_surrogate.py`, as follows:

```
# import libraries
import numpy as np
import csv
import chaospy

# load data
samples = np.loadtxt('samples.txt')
num_samples = len(samples)
```

```
y_vec = []
for i in range(num_samples):
  front_pressure_dir = './checkpoints_flow/network_checkpoint_TL_' + str(i) +'/monitor_domain/results/
     FrontPressure.csv'
  back_pressure_dir = './checkpoints_flow/network_checkpoint_TL_' + str(i) +'/monitor_domain/results/
     BackPressure.csv'
  with open(front_pressure_dir, "r", encoding="utf-8", errors="ignore") as scraped:
    front_pressure = float(scraped.readlines()[-1].split(',')[0])
  with open(back_pressure_dir, "r", encoding="utf-8", errors="ignore") as scraped:
    back_pressure = float(scraped.readlines()[-1].split(',')[0])
  pressure_drop = front_pressure - back_pressure
  y_vec.append(pressure_drop)
y_vec = np.array(y_vec)

# Split data into training and validation
val_portion = 0.16
val_idx = np.random.choice(np.arange(num_samples, dtype=int), int(val_portion*num_samples), replace=False)
val_x, val_y = samples[val_idx], y_vec[val_idx]
train_x, train_y = np.delete(samples, val_idx, axis=0).T, np.delete(y_vec, val_idx).reshape(-1,1)

# Construct the PCE
distribution = chaospy.J(chaospy.Uniform(0., np.pi/6),
                         chaospy.Uniform(0., np.pi/6),
                         chaospy.Uniform(0., np.pi/6),
                         chaospy.Uniform(0., np.pi/6))
expansion = chaospy.generate_expansion(2, distribution)
poly = chaospy.fit_regression(expansion, train_x, train_y)

# PCE closed form
print('__________')
print('PCE closd form:')
print(poly)
print('__________')

# Validation
print('PCE evaluatins:')
for i in range(len(val_x)):
  pred = poly(val_x[i,0],
              val_x[i,1],
              val_x[i,2],
              val_x[i,3])[0]
  print('Sample:', val_x[i])
  print('True val:', val_y[i])
  print('Predicted val:', pred)
  print('Relative error (%):', abs(pred-val_y[i])/val_y[i] * 100)
print('__________')
```

Listing 143: Constructing a gPC surrogate for the pressure drop for a 4-parameter NVSwitch heat sink

Figure 83 shows the gPC surrogate performance on the test set. The relative errors are below 1%, showing the good accuracy of the constructed gPC pressure drop surrogate.

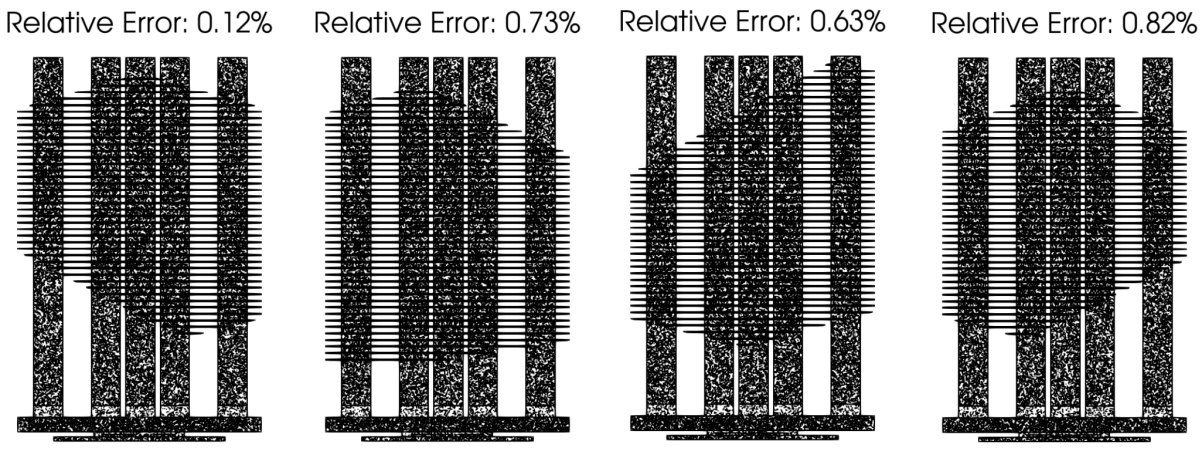


Figure 83: The gPC pressure drop surrogate accuracy tested on four geometries.

19 Case Study: Performance Upgrades and Parallel Processing using Multi-GPU Configurations

19.1 Introduction

In this example we show how to increase the performance of Modulus runs using the advanced features like XLA, TF32, multi-GPU, etc. We will also present some studies that show the scalability of Modulus across multiple GPUs.

19.2 Running jobs using Accelerated Linear Algebra (XLA)

XLA is a domain specific compiler that allows for just-in-time compilation of TensorFlow graphs. PINNs used in Modulus have many peculiarities including the presence of many pointwise operations. Such operations, while being computationally inexpensive, put a large pressure on the memory subsystem of a GPU. XLA allows for kernel fusion, so that many of these operations can be computed simultaneously in a single kernel and thereby reducing the number of memory transfers from GPU memory to the compute units. Kernel fusion using XLA accelerates a single training iteration in Modulus by up to 1.5x.

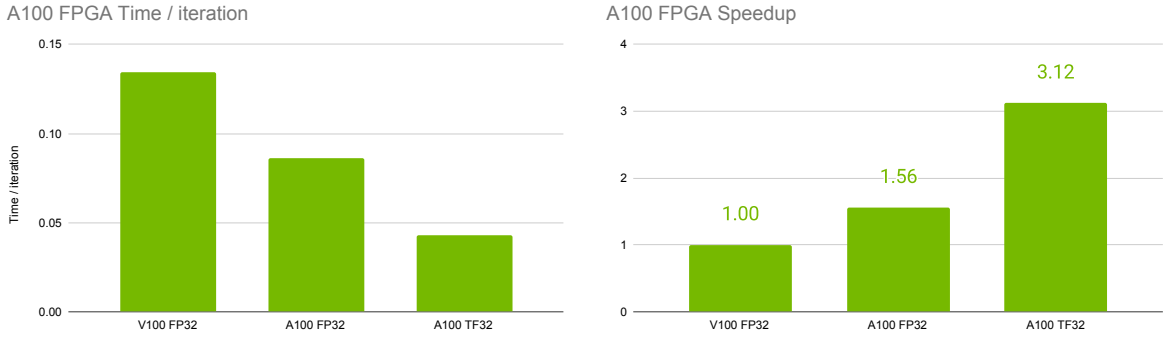
To run a job using XLA, you can pass the `--xla=True` flag while executing the script. This flag can be used for single GPU and multi-GPU runs alike. The following command shows how to run a job using XLA.

```
python fpga_flow_solver.py --xla=True
```

19.3 Running jobs using TF32 math mode

TensorFloat-32 (TF32) is a new math mode available on NVIDIA A100 GPUs for handing matrix math and tensor operations used during the training of a neural network. More details about this feature can be found at https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

On A100 GPUs, the TF32 feature is "ON" by default and we do not need to make any modifications to the regular scripts to use this feature. With this feature, we can obtain up to 1.6x speed-up over FP32 on A100 GPUs and up to 3.1x speed-up over FP32 on V100 GPUs for the FPGA problem. This allows us to achieve same results at a reduced training time as shown in Table 15 and loss convergence plots in Figure 85.



(a) Time per iteration

(b) Speed-up

Figure 84: Accelerated training using TF32 on A100 GPUs

19.4 Running jobs using multiple GPUs

To boost performance and to run larger problems, Modulus supports multi-GPU and multi-node scaling using Horovod. This allows for multiple processes, each targeting a single GPU, to perform independent forward and backward passes and aggregate the gradients collectively before updating the model weights. The Figure 86 shows the scaling performance of Modulus on a annular ring test problem (script can be found at `examples/annular_ring/annular_ring.py`) up to 128 V100 GPUs on 16 nodes. The scaling efficiency from 1 to 32 GPUs is more than 95%.

This data parallel fashion of multi-GPU training keeps the number of points sampled per GPU constant while increasing the total effective batch size. We can use this to our advantage to increase the number of points sampled by increasing the number of GPUs allowing us to handle much larger problems.

Table 15: Comparison of results with and without TF32 math mode

Case Description	P_{drop} (Pa)	Compute Time (hrs)
Modulus: Fully Connected Networks with FP32	29.24	86.9
Modulus: Fully Connected Networks with TF32	29.13	39.5
OpenFOAM Solver	28.03	NA
Commercial Solver	28.38	NA

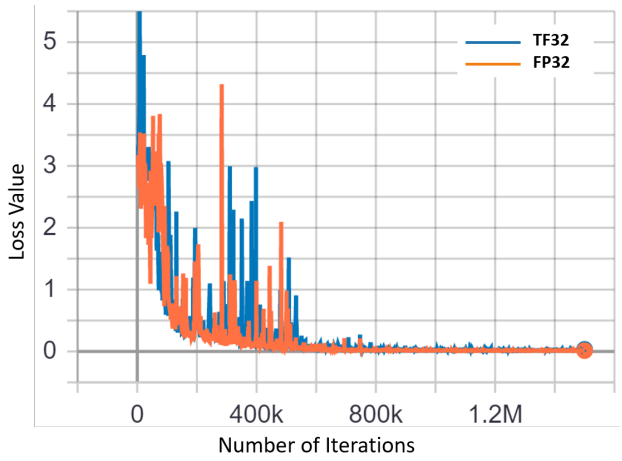


Figure 85: Loss convergence plot for FPGA simulation with TF32 feature

To run a Modulus solution using multiple GPUs on a single compute node, one can first find out the available GPUs using

```
nvidia-smi
```

Once you have found out the available GPUs, you can run the job using `horovodrun -np #GPUs`. Below command shows how to run the job using 2 GPUs.

```
horovodrun -np 2 python fpga_flow_solver.py
```

You can find more information at: https://docs.nvidia.com/deeplearning/frameworks/tensorflow-release-notes/rel_20-07.html#rel_20-07

19.4.1 Automatically increase the learning rate with number of GPUs

In Section 1.6.2.3 of tutorial 1, we describe how the learning rate can be scaled with the batch size to achieve faster time to convergence. As suggested earlier, we can run the same scripts using multi-GPU to achieve larger batch sizes. Doing so, the time per iteration remains fairly constant and the benefits are mostly in terms of large problem solution (Figure 86). However, we can also decrease the total time to convergence by scaling the learning rate linearly with the number of GPUs. As described in [16], simply increasing the learning rate can cause the model to diverge at large batch sizes. This can be fixed by using an initial learning rate warm-up.

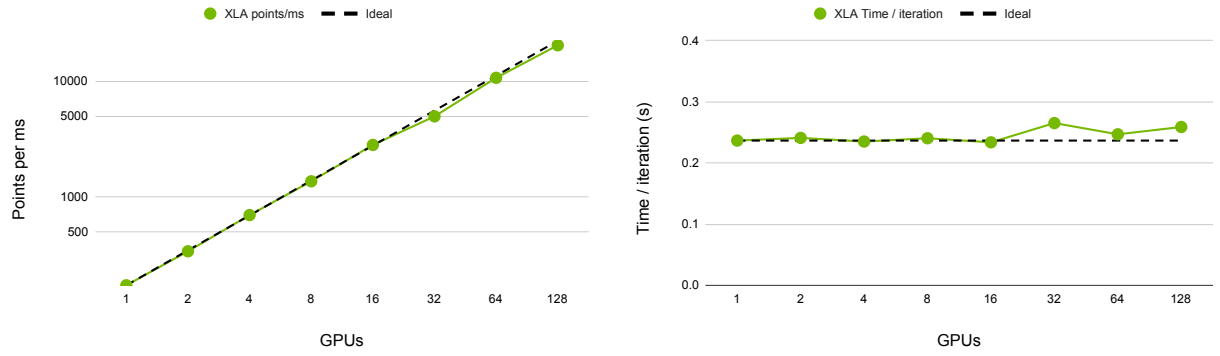
Below are the code changes are required to the script to make use of this feature (with a gradual warm-up) in Modulus.

```
...
from modulus.learning_rate import ExponentialDecayLRWithWarmup

class FPGAFlowSolver(Solver):
  train_domain = FPGAFlowTrain
  monitor_domain = FPGAFlowMonitor
```

```
  val_domain = FPGAFlowVal
  arch = FourierNetArch
  lr = ExponentialDecayLRWithWarmup

  def __init__(self, **config):
    super(FPGAFlowSolver, self).__init__(**config)

    self.frequencies = ('axis,diagonal', [i for i in range(35)])

    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node(stop_gradients=['nu', 'nu__x', '
     nu__y', 'nu__z'])
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_fpga_fluid_flow',
        'rec_results_cpu': True,
        'max_steps': 4000000,
        'decay_steps': 15000,
        'warmup_type': 'gradual',
        'warmup_steps': 30000,
        'rec_results_freq': 5000,
        })

if __name__ == '__main__':
  ctr = ModulusController(FPGAFlowSolver)
  ctr.run()
```

Listing 144: Exponential learning rate decay with warm-up for multi-GPU runs

Figure 86 shows the weak scaling performance of Modulus for the annular ring problem. Modulus scales very well up to 128 GPUs with an almost constant time per iteration. When coupled with the learning rate scaling with warmup as described in Section 1.6.2.3, this can lead to accelerated time to convergence. Figure 87 shows the learning rate schedule and the loss function evolution as the number of GPUs is increased from 1 to 16 for the NVSwitch heatsink case. For the multi-GPU cases, the learning rate is gradually increased from the baseline case and this allows the model to train without diverging early on and allows the model to converge faster as a result of the increased global batch size coupled with the increased learning rate.



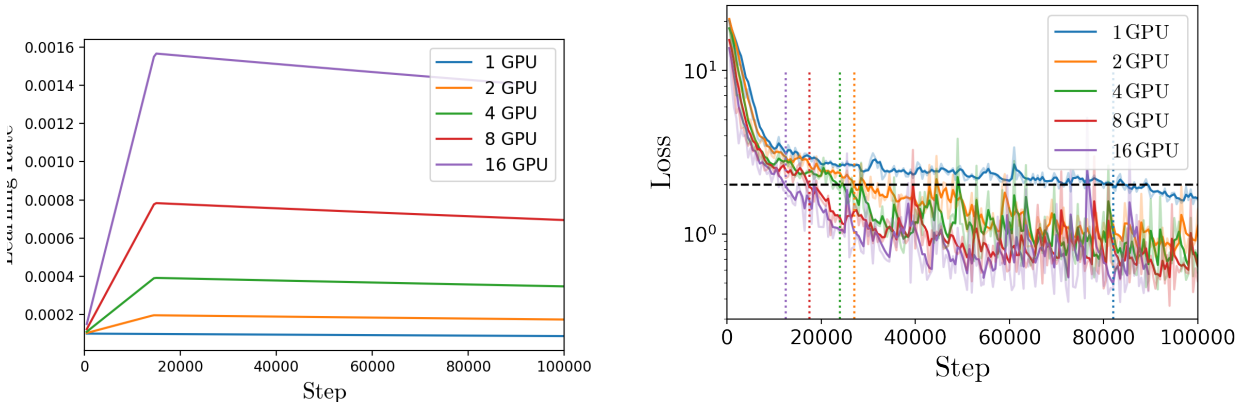
(a) Points per ms

(b) Time per iteration

Figure 86: Modulus weak scaling plots for flow over an annular ring problem

19.4.2 Strong scaling to multiple GPUs for faster time to convergence

Weak scaling with learning rate schedules is an efficient way to get accelerated convergence. But in some cases, large batch training issues might reduce the effectiveness of this approach. Strong scaling the problem is a viable solution in these cases. With strong scaling, a fixed batch size is split across multiple GPUs in order to reduce the wall clock computation time. As the number of GPUs is increased, the global batch size remains constant and the batch size per GPU reduces proportionally. Figure 88 shows the time per iteration for the 3D Taylor-Green vortex problem as the

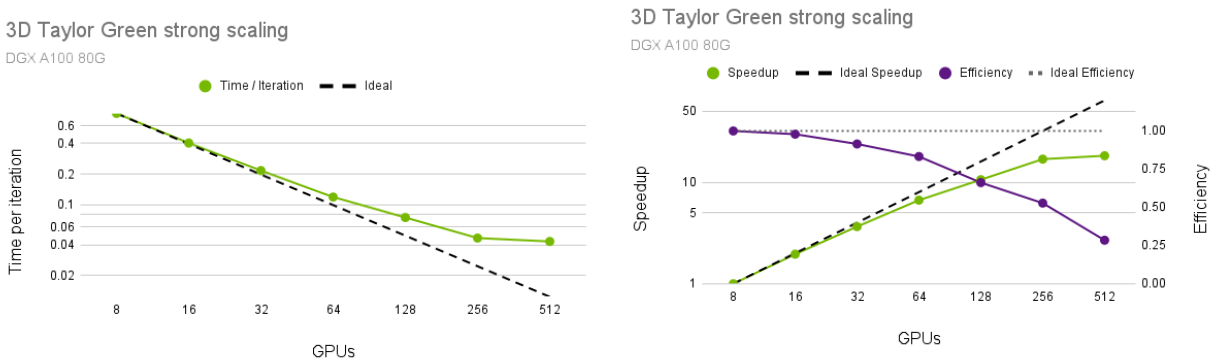


(a) Learning rate

(b) Loss function value

Figure 87: Accelerating time to convergence using multiple GPUs and learning rate schedules.

problem is strong scaled. For this problem, we are able to scale the problem efficiently up to 128 GPUs. We get a further speed-up going to 512 GPUs but with a loss in efficiency due to an increase in communication time while the computation time decreases.



(a) Time per iteration

(b) Speedup (green, left axis) and scaling efficiency (purple, right axis)

Figure 88: Strong scaling results for the Taylor-Green vortex problem on A100 GPUs

References

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[2] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[3] Luning Sun, Han Gao, Shaowu Pan, and Jian-Xun Wang. Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, 361:112732, 2020.

[4] DL Young, CH Tsai, and CS Wu. A novel vector potential formulation of 3d navier–stokes equations with through-flow boundaries by a local meshless method. *Journal of Computational Physics*, 300:219–240, 2015.

[5] Mohammad Amin Nabian, Rini Jasmine Gladstone, and Hadi Meidani. Efficient training of physics-informed neural networks via importance sampling. *arXiv preprint arXiv:2104.12325*, 2021.

[6] John H Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2(1):84–90, 1960.

[7] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310, 2019.

[8] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *arXiv preprint arXiv:2003.08934*, 2020.

[9] Matthew Tancik, Pratul P Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *arXiv preprint arXiv:2006.10739*, 2020.

[10] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient pathologies in physics-informed neural networks. *arXiv preprint arXiv:2001.04536*, 2020.

[11] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[12] Vincent Sitzmann, Julien NP Martel, Alexander W Bergman, David B Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *arXiv preprint arXiv:2006.09661*, 2020.

[13] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

[14] Rizal Fathony, Anit Kumar Sahu, Devin Willmott, and J Zico Kolter. Multiplicative filter networks. 2021.

[15] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7482–7491, 2018.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[17] Ameya D Jagtap, Kenji Kawaguchi, and George Em Karniadakis. Adaptive activation functions accelerate convergence in deep and physics-informed neural networks. *Journal of Computational Physics*, 404:109136, 2020.

[18] Dietrich Braess. *Finite elements: Theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, 2007.

[19] David Gilbarg and Neil S Trudinger. *Elliptic partial differential equations of second order*. springer, 2015.

[20] Lawrence C Evans. Partial differential equations and monge-kantorovich mass transfer. *Current developments in mathematics*, 1997(1):65–126, 1997.

[21] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. Variational physics-informed neural networks for solving partial differential equations. *arXiv preprint arXiv:1912.00873*, 2019.

[22] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. hp-vpinns: Variational physics-informed neural networks with domain decomposition. *arXiv preprint arXiv:2003.05385*, 2020.

[23] Dongbin Xiu. *Numerical methods for stochastic computations: a spectral method approach*. Princeton university press, 2010.

[24] Peter Monk. A finite element method for approximating the time-harmonic maxwell equations. *Numerische mathematik*, 63(1):243–261, 1992.

[25] Yaohua Zang, Gang Bao, Xiaojing Ye, and Haomin Zhou. Weak adversarial networks for high-dimensional partial differential equations. *Journal of Computational Physics*, page 109409, 2020.

[26] Bernardo Cockburn, George E Karniadakis, and Chi-Wang Shu. *Discontinuous Galerkin methods: theory, computation and applications*, volume 11. Springer Science & Business Media, 2012.

[27] Chiyu Max Jiang, Soheil Esmaeilzadeh, Kamyar Azizzadenesheli, Karthik Kashinath, Mustafa Mustafa, Hamdi A Tchelepi, Philip Marcus, Anima Anandkumar, et al. Meshfreeflownet: A physics-constrained deep continuous space-time super-resolution framework. *arXiv preprint arXiv:2005.01463*, 2020.

[28] Zachary C Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*, 2015.

[29] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[30] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3):1165–1187, 2019.

[31] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P. H. J. Kelly, F. J. Herrmann, and G. J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *CoRR*, abs/1807.03032, jul 2018.

[32] Colby L Wight and Jia Zhao. Solving allen-cahn and cahn-hilliard equations using the adaptive physics informed neural networks. *arXiv preprint arXiv:2007.04542*, 2020.

[33] Chengping Rao, Hao Sun, and Yang Liu. Physics informed deep learning for computational elastodynamics without labeled data. *arXiv preprint arXiv:2006.08472*, 2020.

[34] Sebastian Scholl, Bart Janssens, and Tom Verstraete. Stability of static conjugate heat transfer coupling approaches using robin interface conditions. *Computers & Fluids*, 172, 06 2018.

[35] Sifan Wang, Hanwen Wang, and Paris Perdikaris. On the eigenvector bias of fourier feature networks: From regression to solving multi-scale pdes with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 384:113938, 2021.

Alphabetical Index