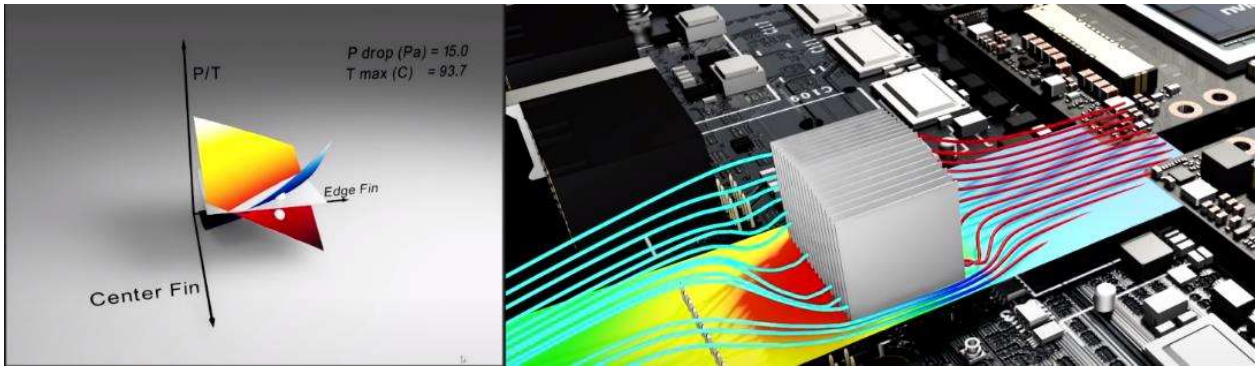
SimNet

A Neural Network Based Partial Differential Equation Solver



User Guide

Release v21.06 | June 7, 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

Getting Started: Guidelines on using this document and System Requirements **8**

Structure of the User Guide . . . 8

What's new in SimNet 21.06 . . . 9

Targeted Users . . . 9

1 Theory **11**

1.1 Introduction . . . 11

1.2 Neural Network Solver Methodology . . . 11

1.3 Physics Informed Neural Networks in SimNet . . . 11

1.3.1 Integral Formulation of losses . . . 11

1.3.2 Integral Equations . . . 12

1.3.3 Parameterized Geometries . . . 13

1.3.4 Inverse Problems . . . 13

1.4 The `examples/` directory . . . 15

1.5 Tips on improving accuracy and convergence speed of PINNs results . . . 16

1.5.1 Integral Continuity Planes . . . 16

1.5.2 Spatial Weighting of Losses (SDF weighting) . . . 16

1.5.3 Increasing the Point cloud density . . . 17

1.5.4 Gradient Aggregation . . . 17

1.5.5 Exact Continuity . . . 19

1.5.6 Importance Sampling . . . 19

1.5.7 Quasi-Random Sampling . . . 19

1.5.8 Symmetry Boundaries . . . 19

1.6 Advanced Schemes and Architectures . . . 22

1.6.1 Network Architectures in SimNet . . . 22

1.6.1.1 Fourier Network . . . 22

1.6.1.2 Modified Fourier Network . . . 22

1.6.1.3 Highway Fourier Network . . . 22

1.6.1.4 SiReNs . . . 23

1.6.1.5 DGM architecture . . . 23

1.6.1.6 Multiplicative Filter Network . . . 23

1.6.2 Other Advanced features in SimNet . . . 24

1.6.2.1 Learning Rate Annealing . . . 24

1.6.2.2 Homoscedastic task uncertainty for loss weighting . . . 24

1.6.2.3 Learning Rate Schedules for Multi-GPU Simulations . . . 24

1.6.2.4 Adaptive Activation Functions . . . 25

1.7 Weak solution of PDEs using PINNs . . . 26

1.8 Generalized Polynomial Chaos . . . 28

Appendices **29**

A Relative Function Spaces and Integral Identities **29**

A.1 L^p space . . . 29

A.2 C^k space . . . 29

A.3 $W^{k,p}$ space . . . 30

A.4 Integral identities . . . 30

B Example: Derivation of Variational form for the interface problem **31**

2 Lid Driven Cavity Flow **33**

2.1 Introduction . . . 33

2.2 Problem Description . . . 33

2.3 Case Setup . . . 33

2.3.1 Creating Geometry . . . 34

2.3.2 Defining the Boundary conditions and Equations to solve . . . 35

2.3.2.1 Boundary conditions: . . . 35

2.3.2.2 Equations to solve: . . . 35

2.3.3 Creating Validation data . . . 37

2.3.4 Making the Neural Network solver . . . 37

2.4 Running the SimNet solver . . . 38

2.5 Results and Post-processing . . . 38

2.5.1 Setting up Tensorboard . . . 38

2.5.2 Trained model . . . 39

3 Turbulent physics: Zero Equation Turbulence Model **42**

3.1 Introduction . . . 42

3.2 Problem Description . . . 42

3.3 Case Setup . . . 42

3.3.1 Creating Geometry and Defining Boundary conditions and Equations . . . 42

3.3.2 Creating Validation data . . . 43

3.3.3 Creating Monitor and Inference domain . . . 43

3.3.3.1 Monitor . . . 43

3.3.3.2 Inference . . . 44

3.3.4 Adding Turbulence Equation and Making the Neural Network solver . . . 44

3.4 Running the SimNet solver . . . 45

3.5 Results and Post-processing . . . 45

3.5.1 Setting up Tensorboard . . . 45

3.5.2 Trained model . . . 45

4 Transient physics: Wave Equation **48**

4.1 Introduction . . . 48
4.2 Problem Description . . . 48
4.3 Writing custom PDEs and boundary/initial conditions . . . 48
4.4 Case Setup . . . 49
4.4.1 Creating Geometry and Defining Initial and Boundary conditions and Equations to solve . . . 50
4.4.2 Creating in Validation data from analytical solutions . . . 50
4.4.3 Making the Neural Network Solver . . . 51
4.5 Running the SimNet solver . . . 51
4.6 Results and Post-processing of non-standard datasets . . . 51
4.7 Temporal loss weighting and time-marching schedule . . . 51
4.8 Experimental RNN and GRU architectures for time-domain problems . . . 52

5 Transient Physics: 2D Seismic Wave Propagation 55
5.1 Introduction . . . 55
5.2 Problem Description . . . 55
5.3 Case Setup . . . 56
5.3.1 Defining the Equations . . . 56
5.3.2 Variable Velocity Model . . . 57
5.3.3 Solving the PDEs: Creating geometry, defining training and validation domains, making the Neural Network solver . . . 57
5.4 Results and Post-processing . . . 59

6 Transient Navier-Stokes via Moving Time Window: Taylor Green Vortex Decay 61
6.1 Introduction . . . 61
6.2 Problem Description . . . 61
6.3 Case Setup . . . 62
6.3.1 Sequence of Train Domains . . . 62
6.3.2 Sequence Solver . . . 64
6.4 Results and Post-processing . . . 66

7 Ordinary Differential Equations: Coupled Spring Mass system 68
7.1 Introduction . . . 68
7.2 Problem Description . . . 68
7.3 Case Setup . . . 68
7.3.1 Defining the Equations . . . 68
7.3.2 Solving the ODEs: Creating Geometry, defining ICs and making the Neural Network Solver . . . 69
7.4 Results and Post-processing . . . 71

8 Scalar Transport: 2D Advection Diffusion 72
8.1 Introduction . . . 72
8.2 Problem Description . . . 72

8.3 Case Setup . . . 72
8.3.1 Creating Geometry . . . 73
8.3.2 Defining the Boundary conditions and Equations to solve . . . 73
8.3.3 Creating Monitors, Inference and Validation domains . . . 74
8.3.4 Making the Neural Network Solver . . . 75
8.4 Running the SimNet solver . . . 76
8.5 Results and Post-processing . . . 76

9 Interface problem by Variational method **78**
9.1 Introduction . . . 78
9.2 Problem Description . . . 78
9.3 Variational Form . . . 79
9.4 Continuous type formulation . . . 79
9.4.1 Creating the Geometry . . . 79
9.4.2 Defining the Boundary conditions and Equations to solve . . . 80
9.4.3 Creating the Validation Domains . . . 80
9.4.4 Creating the Variational Loss and Solver . . . 81
9.4.5 Results and Post-processing . . . 83
9.5 Discontinuous type formulation . . . 83
9.5.1 Defining the Boundary conditions and Equations . . . 84
9.5.2 Creating the Variational Loss and Solver . . . 84
9.5.3 Results and Post-processing . . . 86
9.6 Quadrature . . . 86
9.7 Point source and Dirac Delta function . . . 87
9.7.1 Creating the Geometry . . . 88
9.7.2 Creating the Variational Loss and Solver . . . 89
9.7.3 Results and Post-processing . . . 90

10 Electromagnetics: Frequency Domain Maxwell's Equation **91**
10.1 Introduction . . . 91
10.2 Problem 1: 2D Waveguide Cavity . . . 91
10.2.1 Case Setup . . . 91
10.3 Problem 2: 2D Dielectric slab waveguide . . . 94
10.3.1 Case setup . . . 94
10.3.2 Results . . . 95
10.4 Problem 3: 3D waveguide cavity . . . 95
10.4.1 Problem setup . . . 95
10.4.2 Case setup . . . 96
10.4.3 Results . . . 98
10.5 Problem 4: 3D Dielectric slab waveguide . . . 98

10.5.1 Case setup . . . 99
10.5.2 Results . . . 100

11 Linear Elasticity **102**
11.1 Introduction . . . 102
11.2 Prerequisites . . . 102
11.3 Linear Elasticity in the Differential Form . . . 102
11.3.1 Linear elasticity equations in the displacement form . . . 102
11.3.2 Linear elasticity equations in the mixed form . . . 102
11.3.3 Non-dimensionalized linear elasticity equations . . . 103
11.3.4 Plane stress equations . . . 103
11.3.5 Problem 1: Deflection of a bracket . . . 104
11.3.5.1 Case Setup and Results . . . 104
11.3.6 Problem 2: Stress analysis for aircraft fuselage panel . . . 106
11.3.6.1 Case Setup and Results . . . 106
11.4 Linear Elasticity in the Variational Form . . . 106
11.4.1 Linear elasticity equations in the variational form . . . 106
11.4.2 Problem 3: Plane displacement . . . 106
11.4.2.1 Case Setup and Results . . . 109

12 Tuning Neural Network Hyperparameters & Using SimNet's Advanced Features **113**
12.1 Introduction . . . 113
12.2 Network Architecture . . . 114
12.3 Activation Functions . . . 115
12.4 Learning Rate Schedule . . . 115
12.5 Optimizers . . . 115
12.6 Gradient Aggregation . . . 116
12.7 Importance Sampling . . . 117
12.8 Quasirandom Sampling . . . 120
12.9 Example: Radial Basis Neural Networks . . . 120

13 Multi-Physics Simulations: Conjugate Heat Transfer **123**
13.1 Introduction . . . 123
13.2 Problem Description . . . 123
13.3 Case Setup . . . 123
13.3.1 Creating Geometry . . . 124
13.3.1.1 Varying point sampling density & adding parameterized integral continuity planes . 125
13.3.2 Defining the Flow Boundary conditions and Equations . . . 126
13.3.3 Defining the Thermal Multi-Phase Boundary conditions and Equations . . . 127
13.3.4 Creating Validation and Monitor domains . . . 129
13.4 Making the neural network, Multi-Phase training . . . 130

13.5 Running the SimNet Solver . . . 131
13.6 Results and Post-processing . . . 131
13.6.1 Plotting gradient quantities: Wall Velocity Gradients . . . 131

14 Forward simulation using STL geometry: Blood Flow in Intracranial Aneurysm 134
14.1 Introduction . . . 134
14.2 Problem Description . . . 134
14.3 Case Setup . . . 135
14.3.1 Using STL files to create Train domain . . . 135
14.3.2 Creating Validation and Monitor domains . . . 137
14.3.3 Making the Neural Network solver . . . 137
14.4 Running the SimNet solver . . . 137
14.5 Results and Post-processing . . . 138
14.6 Accelerating the Training of Neural Network Solvers via Transfer Learning . . . 140

15 Inverse problem: Finding unknown coefficients of a PDE 141
15.1 Introduction . . . 141
15.2 Problem Description . . . 141
15.3 Case Setup . . . 141
15.3.1 Assimilating data from CSV files/point clouds to create Training data . . . 142
15.3.2 Creating Monitor and Inference domain . . . 143
15.3.3 Making the Neural Network Solver for a Inverse problem . . . 143
15.4 Running the SimNet solver . . . 144
15.5 Results and Post-processing . . . 144

16 Parameterized Simulations and Design Optimization: 3D heat sink 146
16.1 Introduction . . . 146
16.2 Problem Description . . . 146
16.3 Case Setup . . . 147
16.3.1 Creating the Parameterized Geometry . . . 147
16.3.2 Defining the Boundary conditions and Equations for a parameterized problem . . . 149
16.3.3 Creating Validation, Monitor and Inference domain for a parameterized simulation . . . 150
16.3.4 Making the Neural Network Solver for a parameterized problem . . . 151
16.4 Running the SimNet solver . . . 152
16.5 Design Optimization . . . 152
16.6 Results and Post-processing . . . 155

17 Case Study: FPGA Heat Sink with Laminar Flow for Single Geometry: Comparisons of Network Architectures, Optimizers and other schemes in SimNet 157
17.1 Introduction . . . 157
17.2 Problem Description . . . 157

17.3 Solver using Fourier Network Architecture . . . 158
17.4 Leveraging Symmetry of the Problem . . . 159
17.5 Imposing Exact Continuity . . . 160
17.6 Results, Comparisons, and Summary . . . 161

18 Industrial Heat Sink simulations **164**
18.1 Introduction . . . 164
18.2 Problem Description . . . 164
18.3 Case Setup . . . 164
18.3.1 Defining Domain . . . 164
18.3.2 Sequence Solver . . . 168
18.3.3 Mesh Grid Evaluation Script . . . 170
18.4 Results and Post-processing . . . 171
18.5 gPC-Based Surrogate Modeling Accelerated via Transfer Learning . . . 172

19 Case Study: Performance Upgrades and Parallel Processing using Multi-GPU Configurations **176**
19.1 Introduction . . . 176
19.2 Running jobs using Accelerated Linear Algebra (XLA) . . . 176
19.3 Running jobs using TF32 math mode . . . 176
19.4 Running jobs using multiple GPUs . . . 176
19.4.1 Automatically increase the learning rate with number of GPUs . . . 177
19.4.2 Strong scaling to multiple GPUs for faster time to convergence . . . 178

Alphabetical Index **182**

SIMNET 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 SimNet. It outlines the PINNs approach of solving PDEs (Partial Differential Equations) and feature improvements in SimNet over the standard PINNs.

Tutorials 2 and 3, on Lid Driven Cavity flow, are intended to give you a thorough description of the SimNet user interface. Having solved this canonical problem, you can easily draw parallels between other PDE solvers and SimNet 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 SimNet. 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 SimNet's sequential solver and solves the canonical Taylor-Green vortex decay problem using the moving time window approach. Tutorial 7 shows the use of SimNet 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 SimNet. 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 SimNet. In this tutorial, SimNet 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 SimNet 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 SimNet. 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 SimNet library.

What's new in SimNet 21.06

SimNet 21.06 consists of several enhancements in terms of new features, physics and solution methodologies. A summary of the additions and changes compared to SimNet 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 SimNet 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 SimNet 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 SimNet 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 SimNet. 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 SimNet 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 SimNet 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: simnet-team@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 SimNet. This container comes with all the prerequisites and dependencies and allows you to get started efficiently with SimNet.

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 SimNet

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 SimNet

SimNet 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 SimNet, we take a different approach and view the losses as integrals. You can instead write $L_{residual}$ in the form,

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

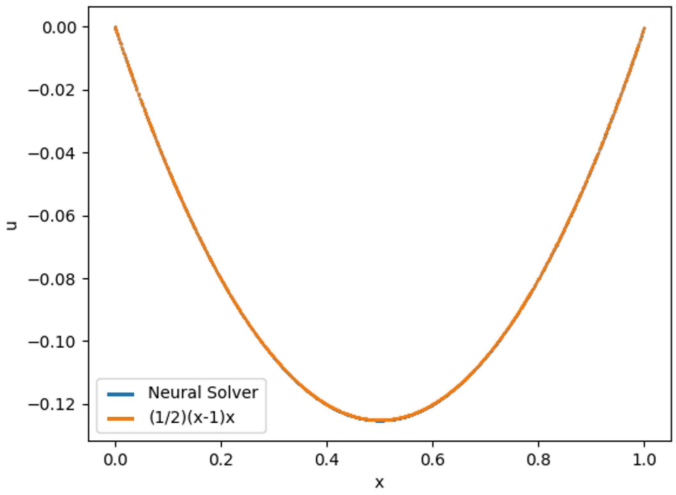


Figure 1: Neural Network Solver compared with analytical solution.

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,

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

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

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.

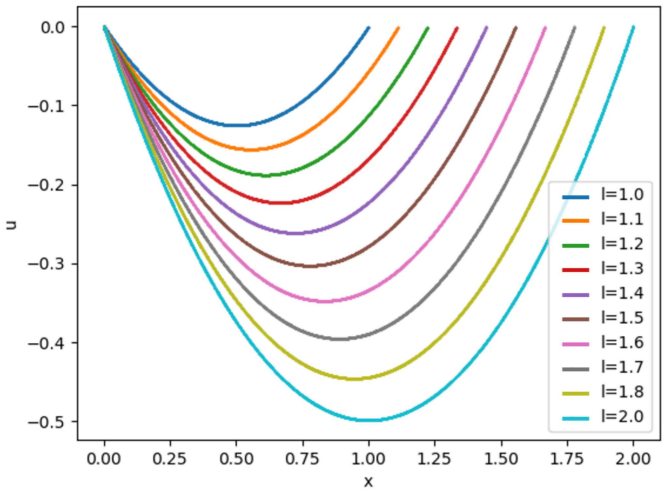


Figure 2: SimNet solving parameterized differential equation problem.

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

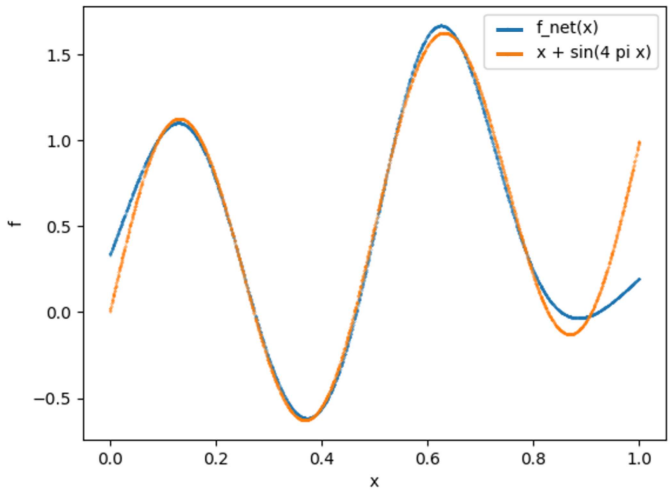


Figure 3: Comparison of true solution for $f(x)$ and the function approximated by the NN.

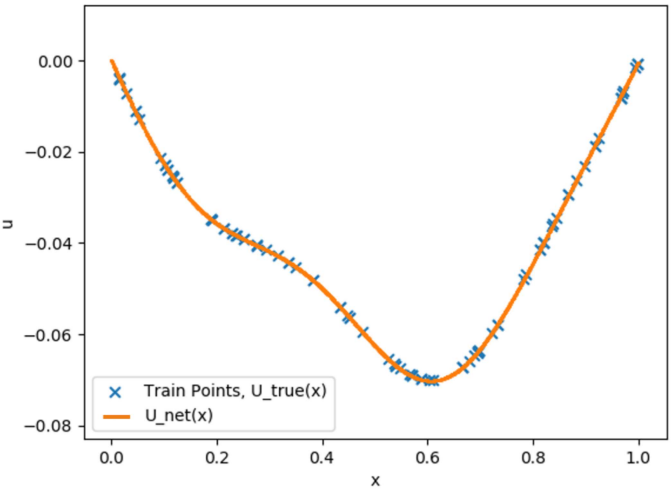


Figure 4: Comparison of $u_{net}(x)$ and train points from u_{true}.

1.4 The examples/ directory

With the SimNet package, we include a wide array of pre-tested examples, and implementations of the simnet 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 SimNet 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 SimNet's 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 simnet 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 SimNet 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.

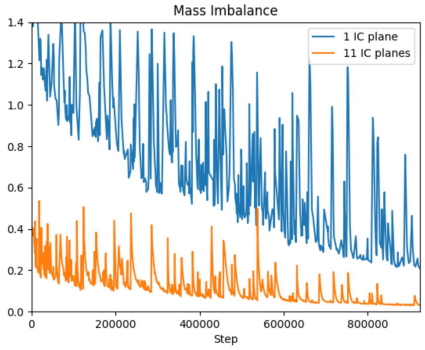


Figure 5: Improvements in accuracy by adding more Integral continuity planes and points inside the domain

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.

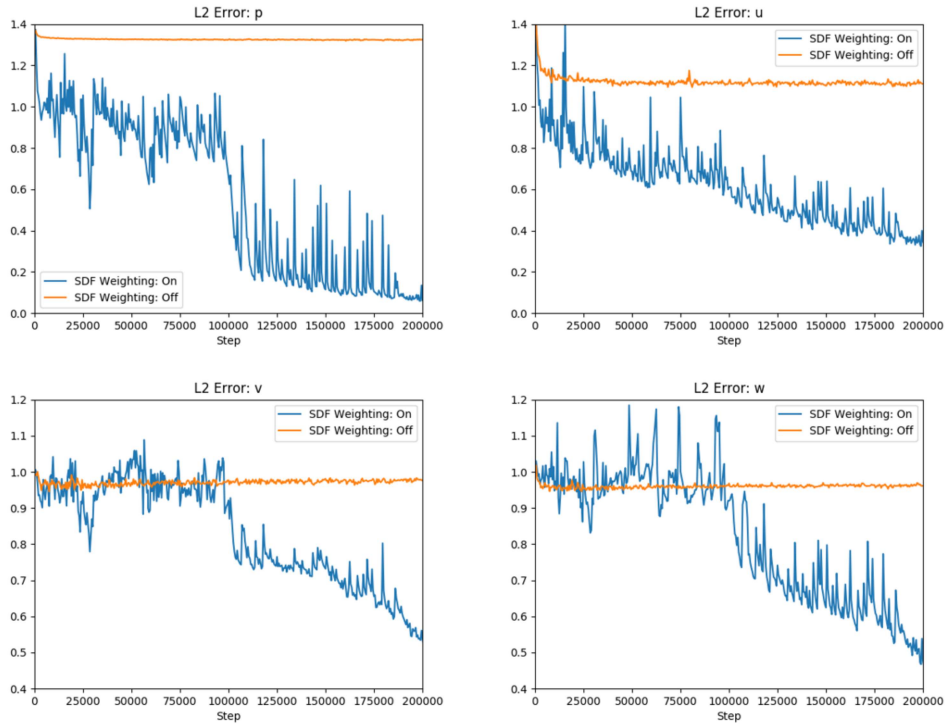


Figure 6: Improvements in convergence speed by weighting the equation residuals spatially.

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 8 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

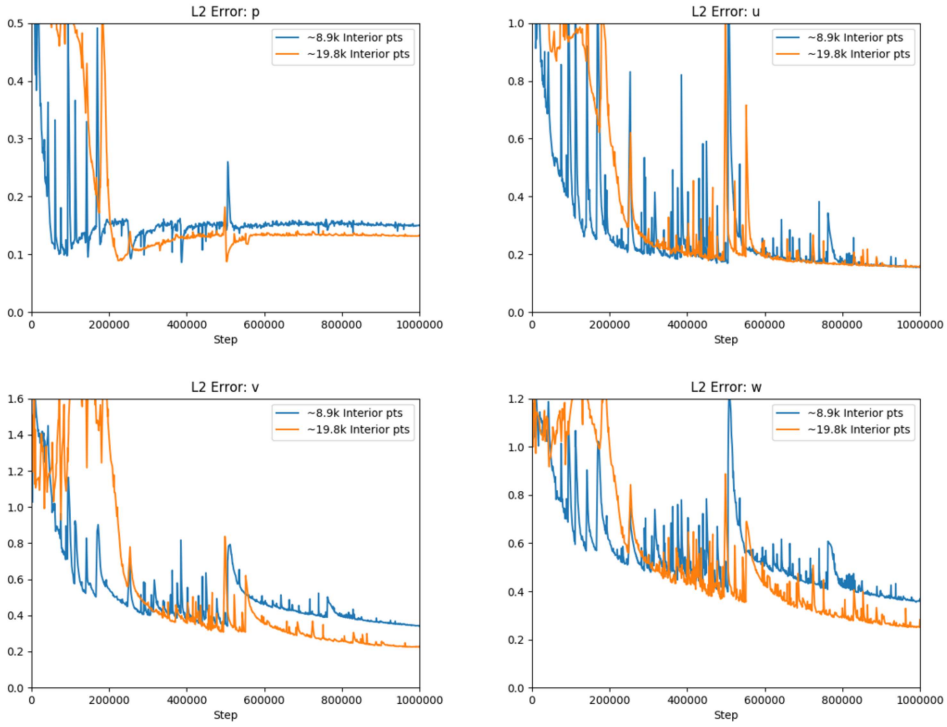


Figure 7: Improvements in accuracy by adding more points in the interior.

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 SimNet is provided in Tutorial 12.

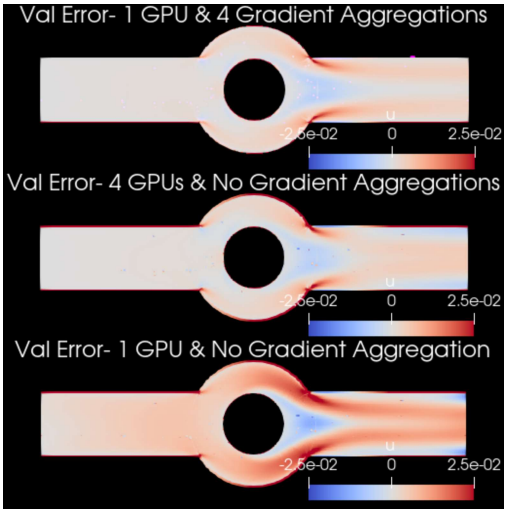


Figure 8: Increasing the batch size can improve the accuracy of neural network solvers. Results are for the u-velocity of an annular ring example trained with different number of GPUs and gradient aggregations.

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

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

SimNet 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 SimNet 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 SimNet 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 SimNet. 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 SimNet, the following symmetry boundary conditions at the line or plane of symmetry may be used:

- Zero value for the physical variables with odd symmetry.

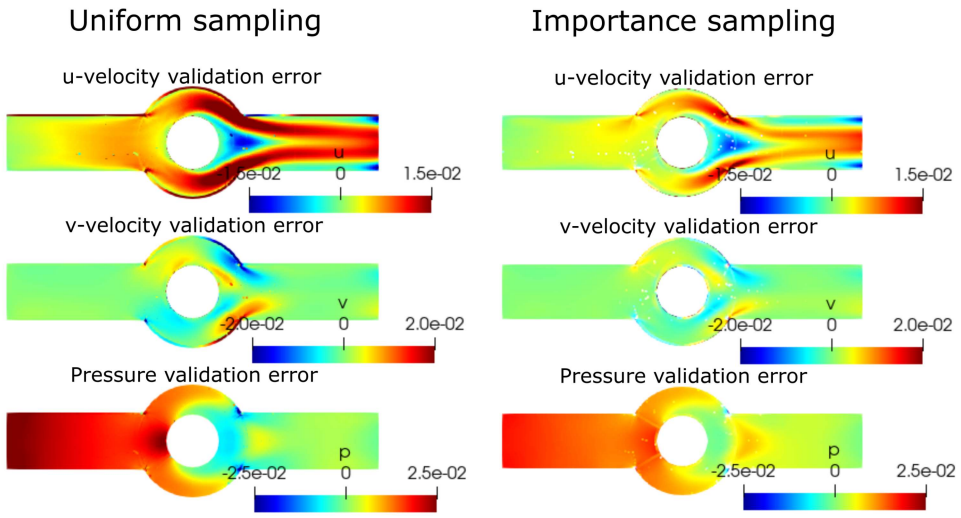


Figure 9: A comparison between the uniform and importance sampling validation error results for the annular ring example.

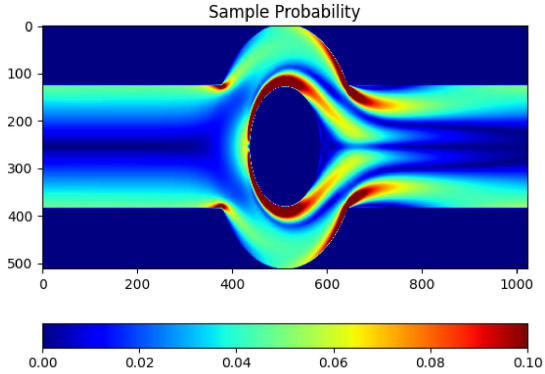


Figure 10: A visualization of the training point sampling probability at iteration 100K for the annular ring example.

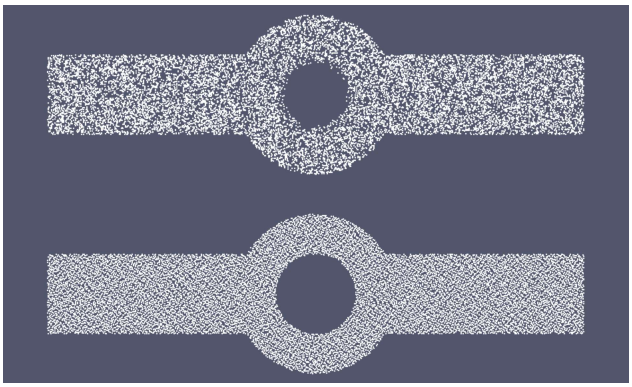


Figure 11: A snapshot of a batch of training points generated using uniform sampling (top) and Halton sequences (bottom) for the annular ring example.

- 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 SimNet library.

1.6.1 Network Architectures in SimNet

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 SimNet 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 SimNet 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 SimNet 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 SimNet's 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 SimNet as well. Details on how to use the SiReN architecture in SimNet 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 SimNet, 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 SimNet

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 SimNet 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

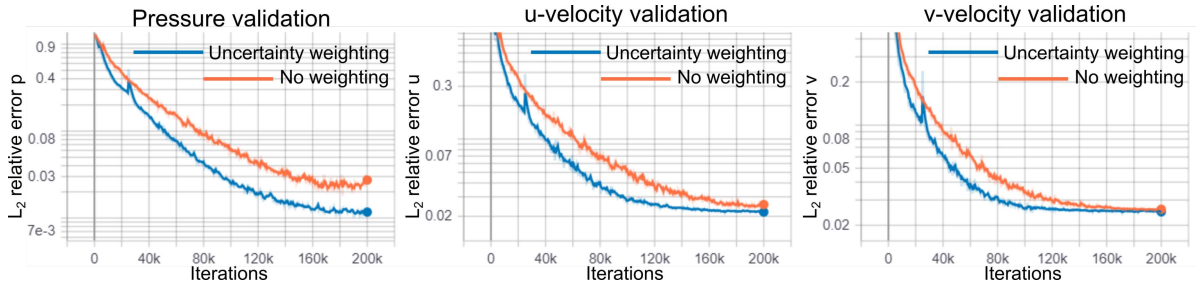


Figure 12: A comparison between the uncertainty loss weighting vs. no loss weighting for the annular ring example.

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. SimNet 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 SimNet 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 SimNet, 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].

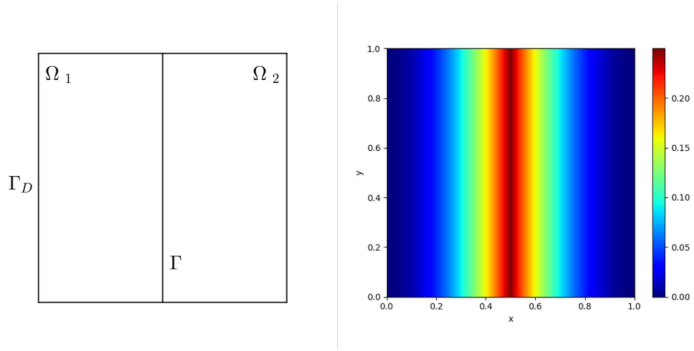


Figure 13: Left: Domain of interface problem. Right: True Solution

The PDEs for the problem are defined as

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

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

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

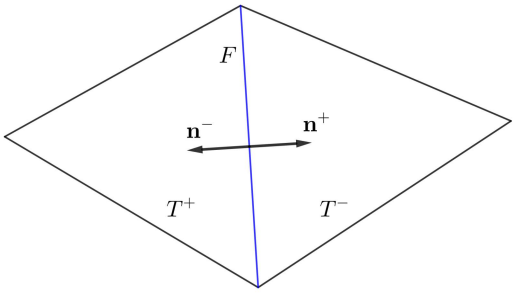


Figure 14: Adjacent Elements.

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 SimNet. In this tutorial, you would learn the following:

1. How to generate a 2D geometry using the geometry module in SimNet.
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 SimNet.

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 SimNet 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.

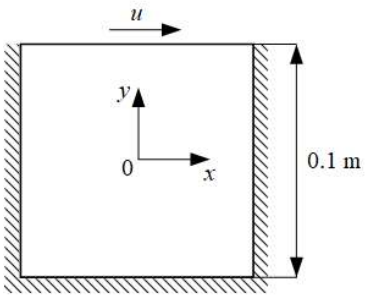


Figure 15: Lid driven cavity geometry

2.3 Case Setup

To set up a case in SimNet, you can create a single python script containing the geometry, boundary conditions, and network information. Once the SimNet 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 SimNet, lets try to summarize the concepts discussed in the Theory chapter and how it relates to SimNet's 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 SimNet, we have modules/functions to help in each stage of this problem definition and solution. The domain can be defined using either the SimNet's 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.

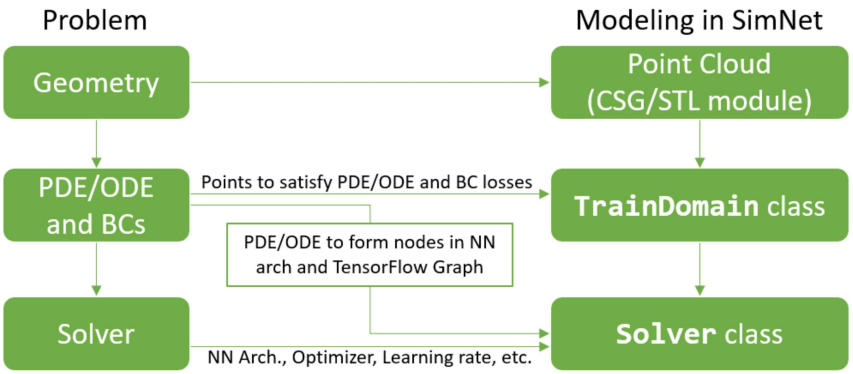


Figure 16: SimNet problem solving methodology

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

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 SimNet. 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 SimNet, 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 SimNet.

```
# 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 SimNet. 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 SimNet, 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 SimNet 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.

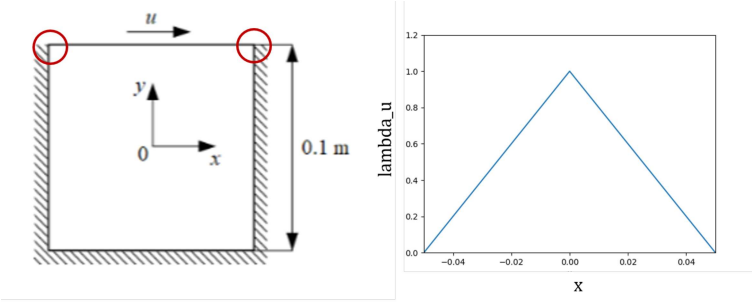


Figure 17: Weighting the sharp discontinuities in the boundary condition

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 SimNet source or refer the source code documentation.

`simnet/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 SimNet 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 SimNet's results. We will now see how to set up such a validation domain in SimNet. We would use the results from OpenFOAM, an open-source CFD solver to make comparisons. The results can be imported into SimNet 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 = SimNetController(LDCSolver)
  ctr.run()
```

Listing 6: Defining the Neural Network Solver

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

2.4 Running the SimNet 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).

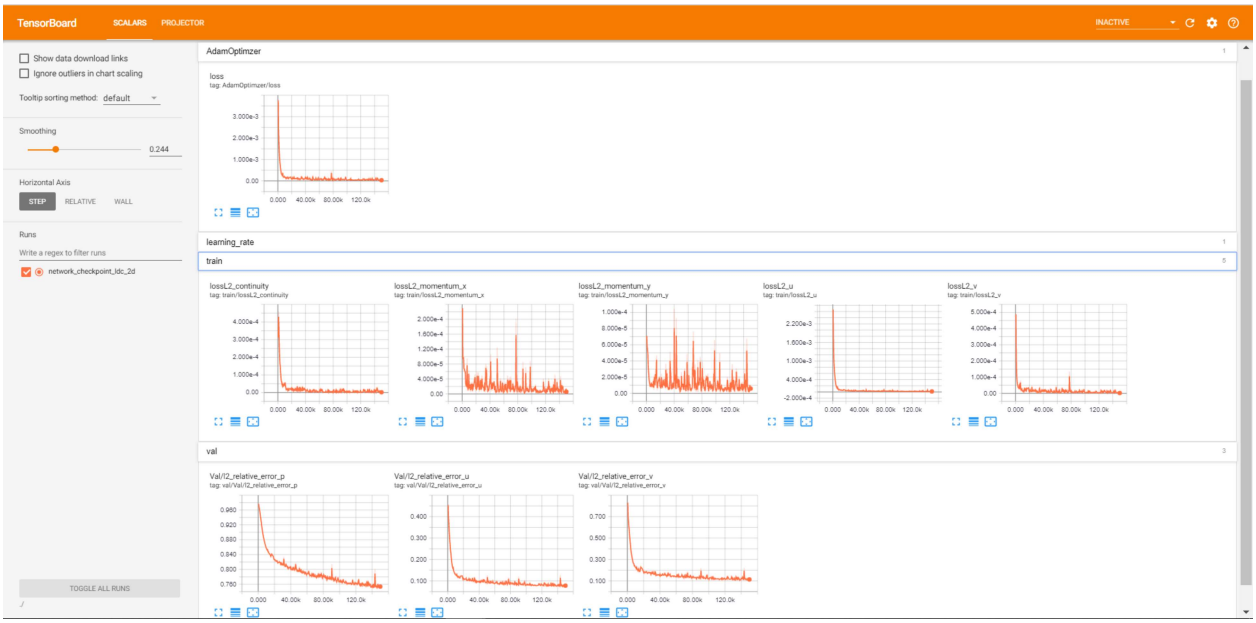


Figure 18: Tensorboard Interface.

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.

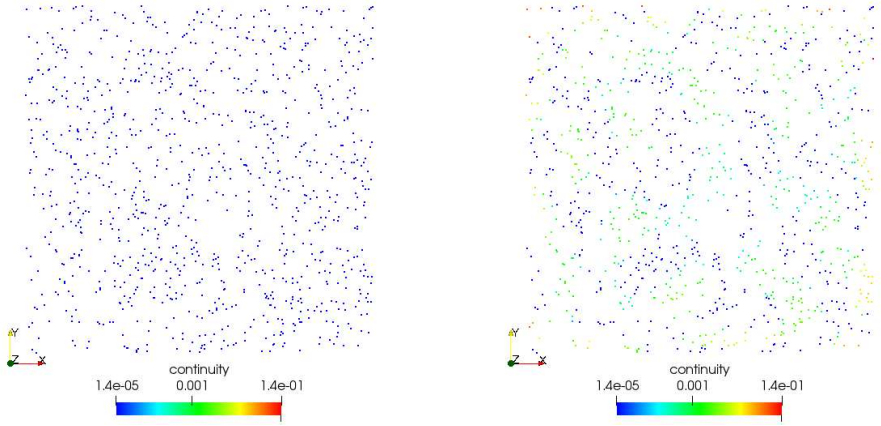


Figure 19: Visualization using Paraview. Left: Continuity as specified in the domain definition. Right: Computed continuity after training.

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 SimNet 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.

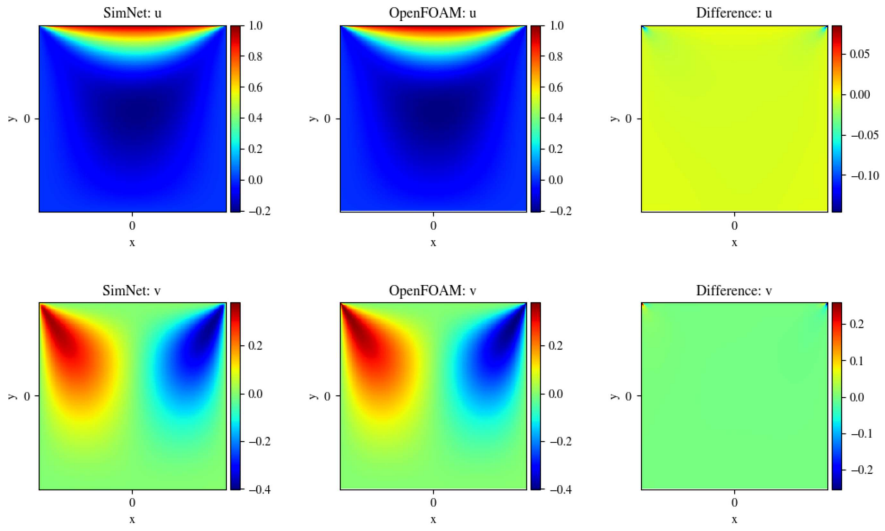


Figure 20: Visualization using custom plot functions. Left: SimNet Prediction, Center: True/Validation Data, Right: Difference

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 SimNet after training	SimNet 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! SimNet 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 SimNet simulations. In this tutorial you would learn the following:

1. How to implement the Zero equation turbulence model in SimNet.
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 SimNet 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 simnet.solver import Solver
from simnet.dataset import TrainDomain, ValidationDomain, MonitorDomain, InferenceDomain
from simnet.data import Validation, Monitor, Inference
from simnet.sympy_utils.geometry_2d import Rectangle
from simnet.csv_utils.csv_rw import csv_to_dict
from simnet.PDES import NavierStokes, ZeroEquation
from simnet.node import Node
from simnet.controller import SimNetController
```

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 SimNet 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 SimNet

Between this and the previous tutorials, we have covered all the available domain classes in SimNet. 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 SimNet 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 = SimNetController(LDCSolver)
  ctr.run()
```

Listing 12: Defining the Neural Network Solver

3.4 Running the SimNet 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`.

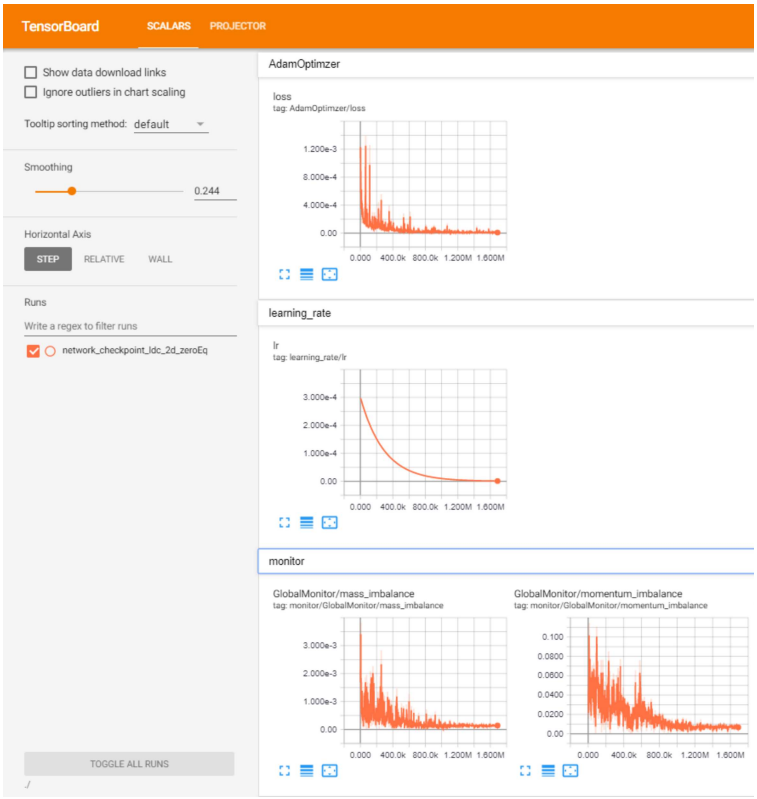


Figure 21: Visualizing Monitors in Tensorboard

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

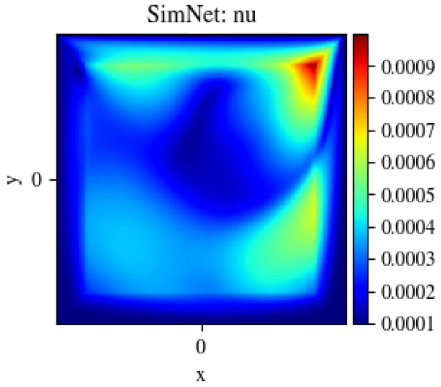


Figure 22: Visualizing variables from Inference domain

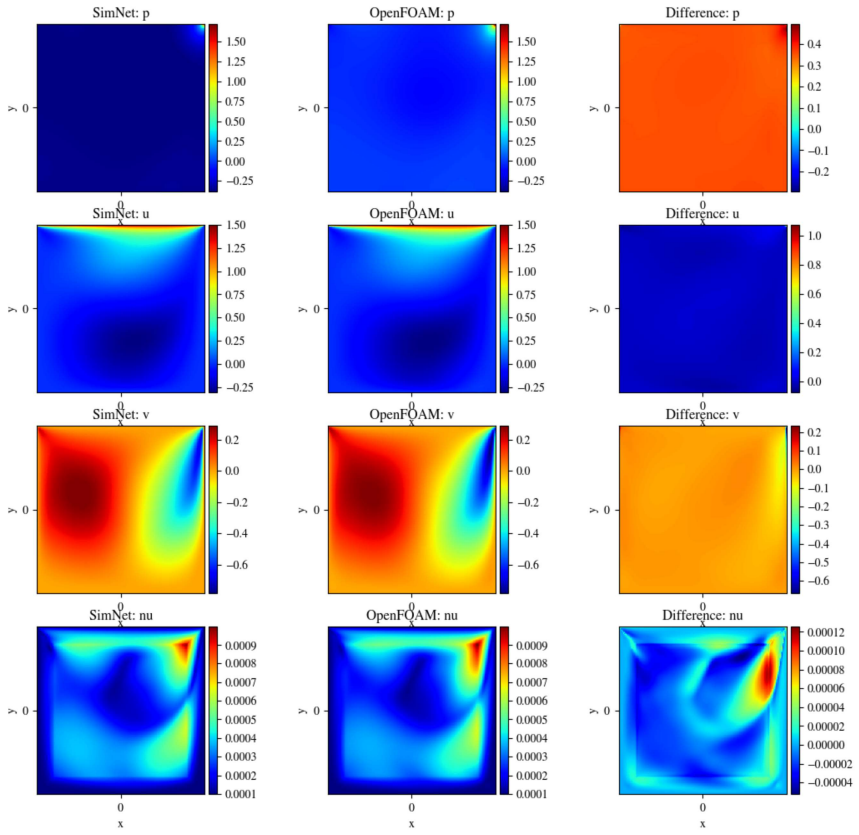


Figure 23: Comparison with OpenFOAM data. Left: SimNet Prediction. Center: OpenFOAM, Right: Difference

4 Transient physics: Wave Equation

4.1 Introduction

In this tutorial, we will walk through the process of setting up a custom PDE in SimNet. 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 SimNet. In this tutorial you would learn the following:

1. How to write your own Partial Differential Equation and boundary conditions in SimNet.
2. How to solve a time-dependent problem in SimNet.
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 SimNet 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 `simnet/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 simnet.pdes import PDES
from simnet.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 SimNet! To verify the implementation, you can refer to the file `simnet/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 simnet.solver import Solver
from simnet.dataset import TrainDomain, ValidationDomain
from simnet.data import Validation
from simnet.sympy_utils.geometry_1d import Line1D
from simnet.controller import SimNetController

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 = SimNetController(WaveSolver)
  ctr.run()
```

Listing 18: Defining the Neural Network Solver

4.5 Running the SimNet 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.

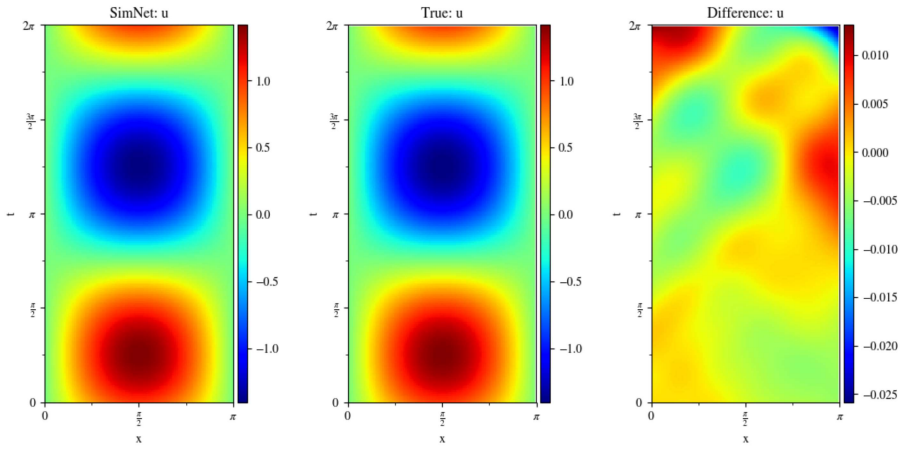


Figure 24: Left: SimNet. Center: Analytical Solution. Right: Difference

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 $[0, T_v(s)]$.

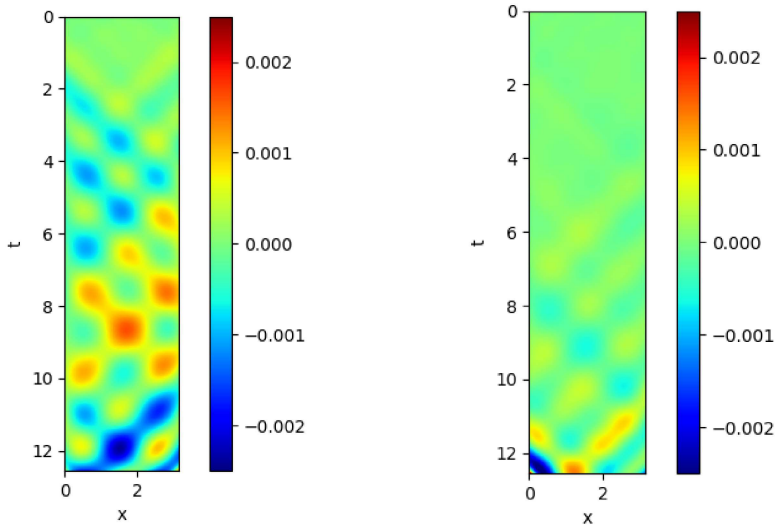
Figures 25, 26, 27 show the SimNet validation error for models trained with and without using temporal loss weighting and time marching for transient 1D and 2D wave examples and a 2D channel flow over a bump. It is evident that these two simple tricks can improve the training accuracy.

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.

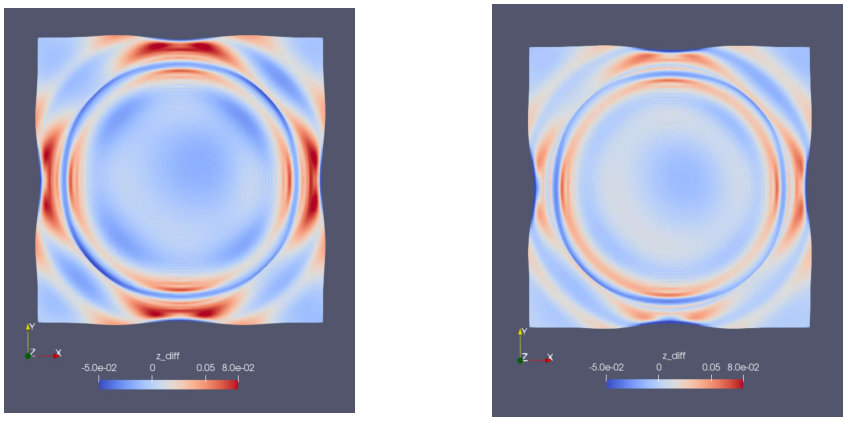
These experimental architectures can be found in `examples/wave_equation/` directory.



(a) Continuous time without temporal weighting and time marching

(b) Continuous time with temporal weighting and time marching

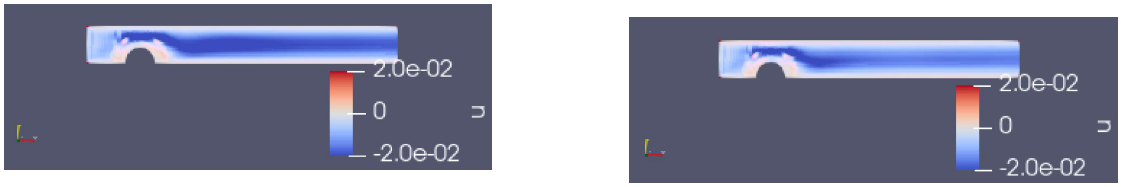
Figure 25: SimNet validation error for the 1D transient wave example: (a) standard continuous time approach; (b) continuous time approach with temporal loss weighting and time marching.



(a) Continuous time without temporal weighting and time marching

(b) Continuous time with temporal weighting and time marching

Figure 26: SimNet 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: SimNet validation error for a 2D transient channel flow over a bump: (a) standard continuous time approach; (b) continuous time approach with temporal loss weighting.

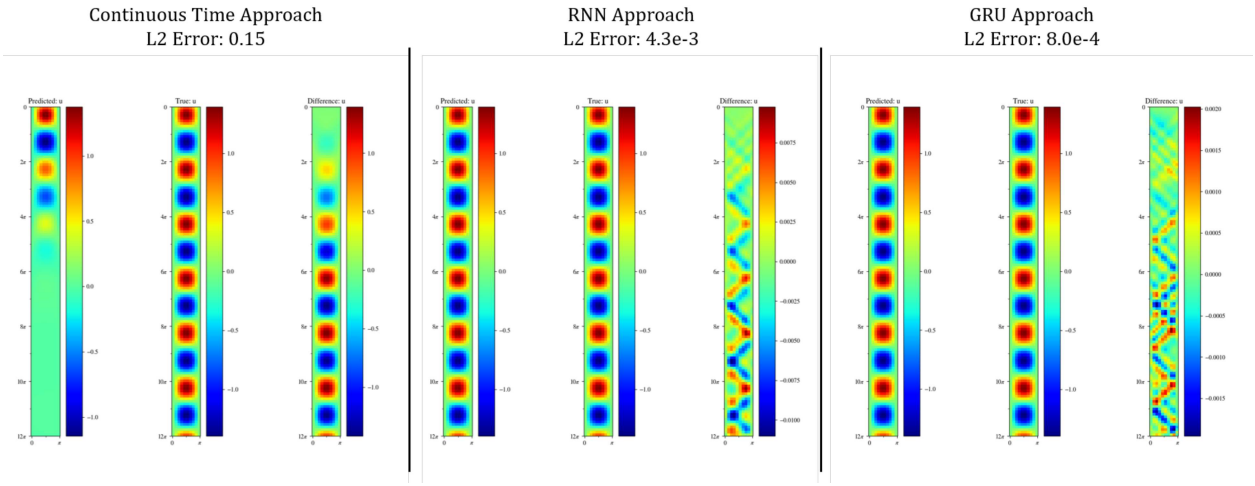


Figure 28: Comparisons of the new experimental architectures vs. continuous time approach

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 SimNet
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 SimNet user interface. Also, we recommend you refer to tutorial 4 for information on defining new differential equations, and solving time-dependent problems in SimNet.

Seismic survey modelling workflow

In this example we aim to use SimNet 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:

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

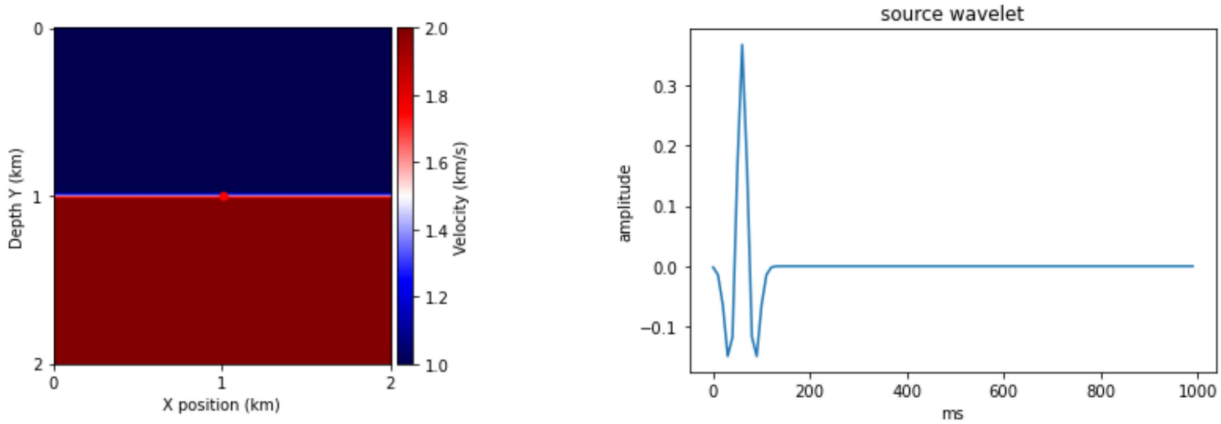
Where $u(\mathbf{x}, t)$ represents the pressure response (known as "wavefield") at location vector $\mathbf{x}$ and time t in an acoustic medium. The problem is solved with zero initial conditions to guarantee unicity of the solution Despite its linearity, the wave equation is notoriously challenging 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 SimNet 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 = SimNetController(WaveSolver)
  ctr.run()
```

Listing 24: Defining equations and making the Neural Network solver

5.4 Results and Post-processing

The results from SimNet 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 SimNet results are noticeably better than Devito, predicting wavefield with much less boundary reflection, specially at later time steps.

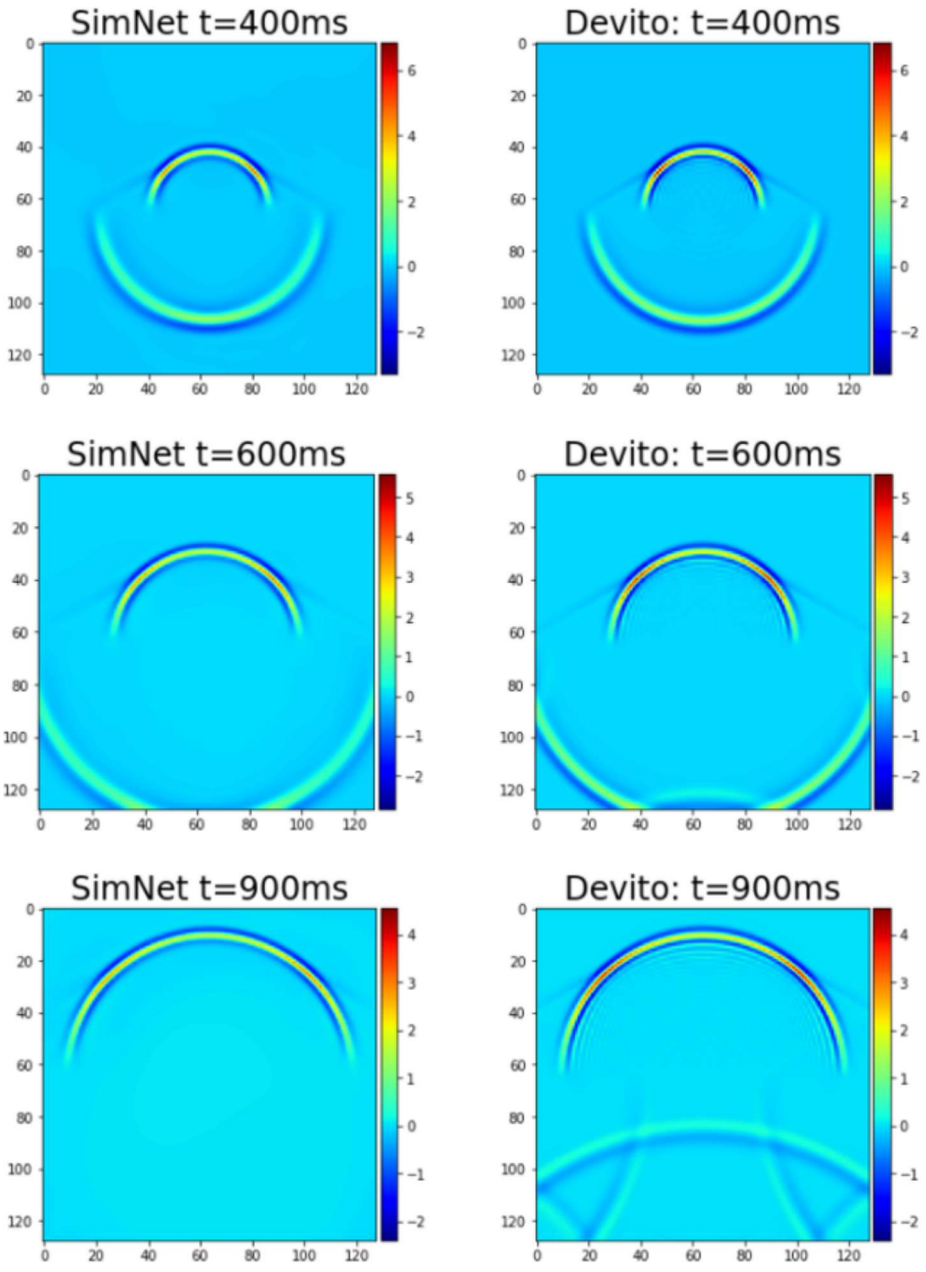


Figure 30: Comparison of SimNet results with Devito solution

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 SimNet
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.

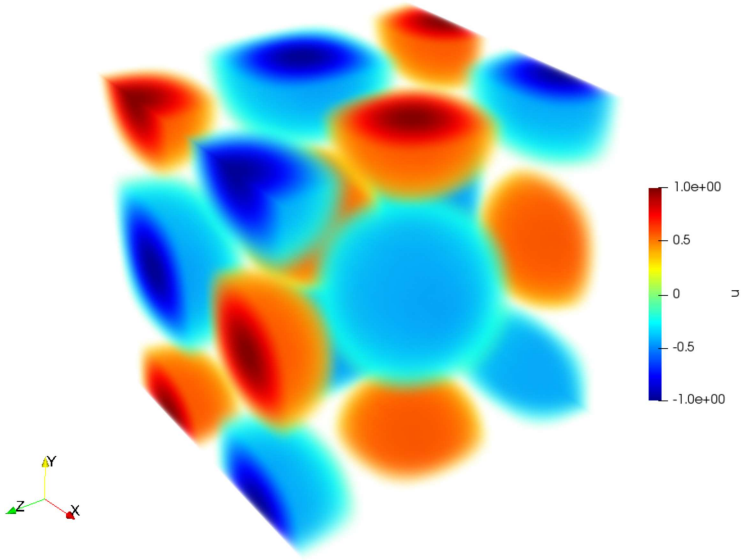


Figure 31: Taylor-Green vortex initial conditions.

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].

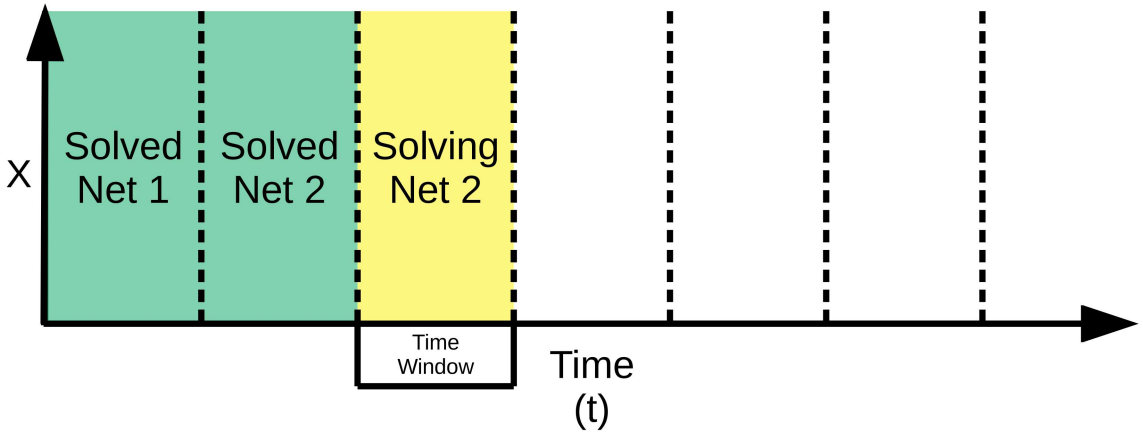


Figure 32: Moving Time Window Method.

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 SimNet 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 SimNet 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 = SimNetController(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.

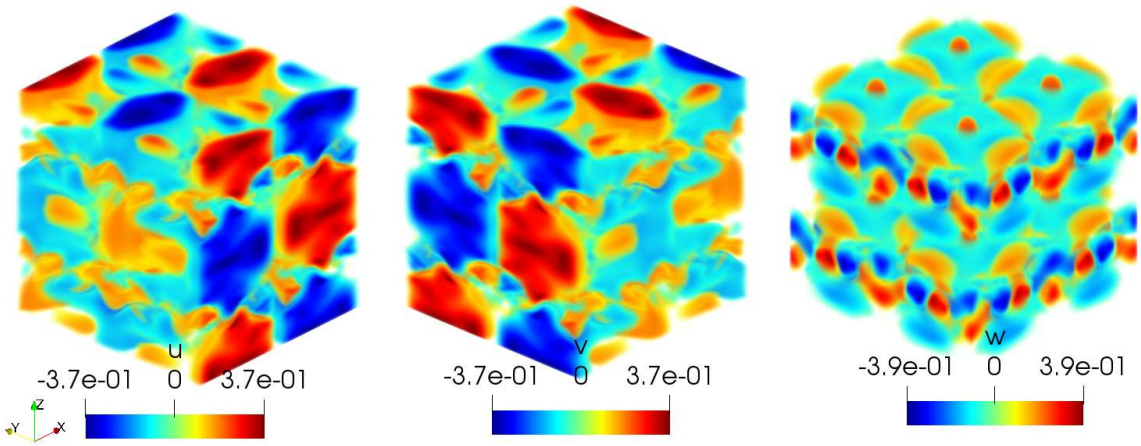


Figure 33: Taylor-Green vortex at time 15.0.

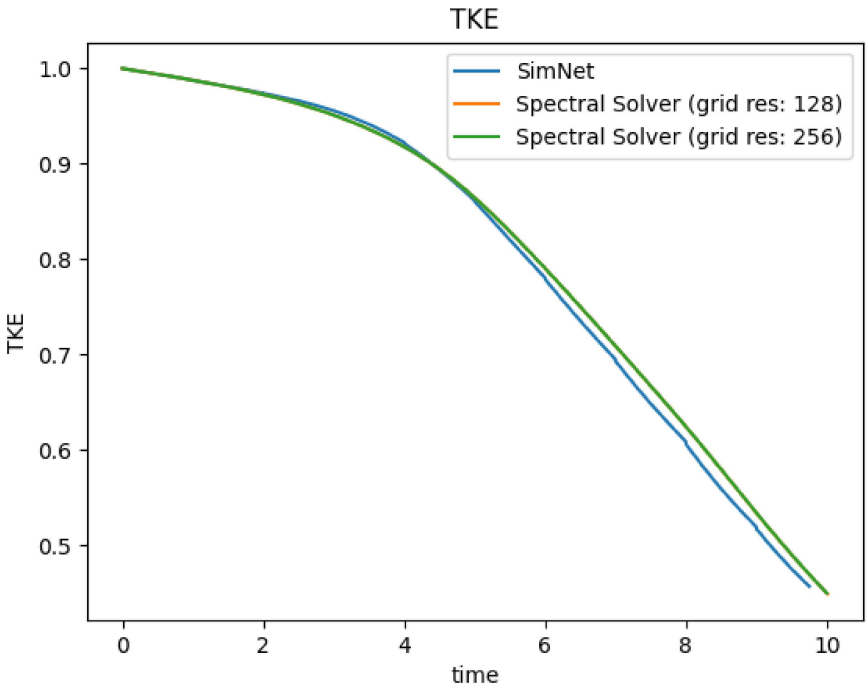


Figure 34: Taylor-Green Trubulent kinetic energy decay.

7 Ordinary Differential Equations: Coupled Spring Mass system

7.1 Introduction

In this tutorial we will use SimNet 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 SimNet user interface. Also, we recommend you to refer to tutorial 4 for information on defining new differential equations, and solving time dependent problems in SimNet.

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:

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

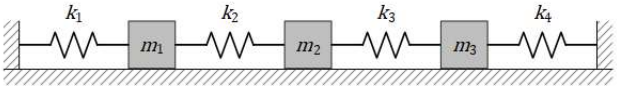


Figure 35: Three masses connected by four springs on a friction-less surface

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 simnet.pdes import PDES
from simnet.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 simnet.solver import Solver
from simnet.dataset import TrainDomain, ValidationDomain
from simnet.data import Validation
from simnet.sympy_utils.geometry_1d import Point1D
from simnet.controller import SimNetController
```

```
from simnet.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 = SimNetController(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 SimNet 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.

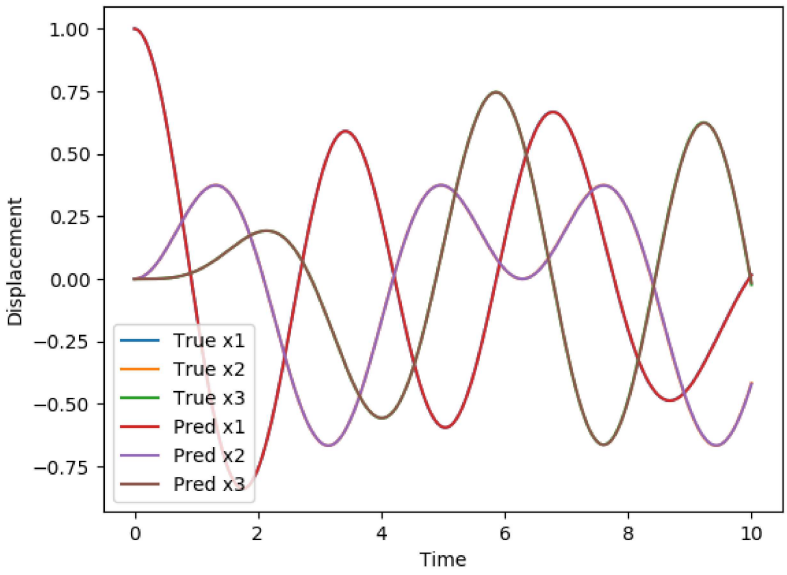


Figure 36: Comparison of SimNet results with analytical solution

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 SimNet 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.

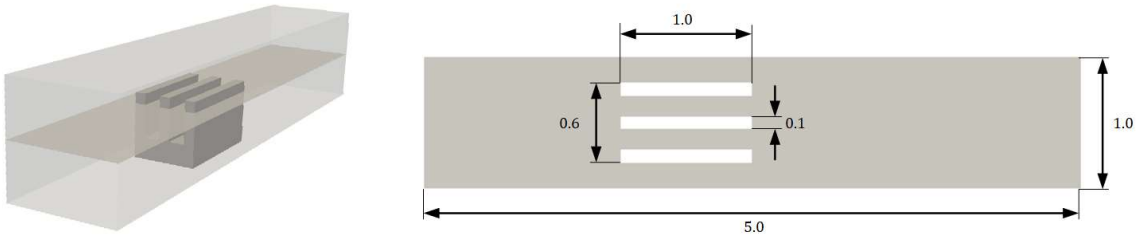


Figure 37: 2d slice of three fin heat sink geometry (All dimensions in m)

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

```
from simnet.PDES.advection_diffusion import AdvectionDiffusion
from simnet.node import Node
from simnet.controller import SimNetController
```

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 = SimNetController(HeatSinkSolver)
  ctr.run()
```

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

8.4 Running the SimNet 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 SimNet simulation are compared against the OpenFOAM data in figure 38.

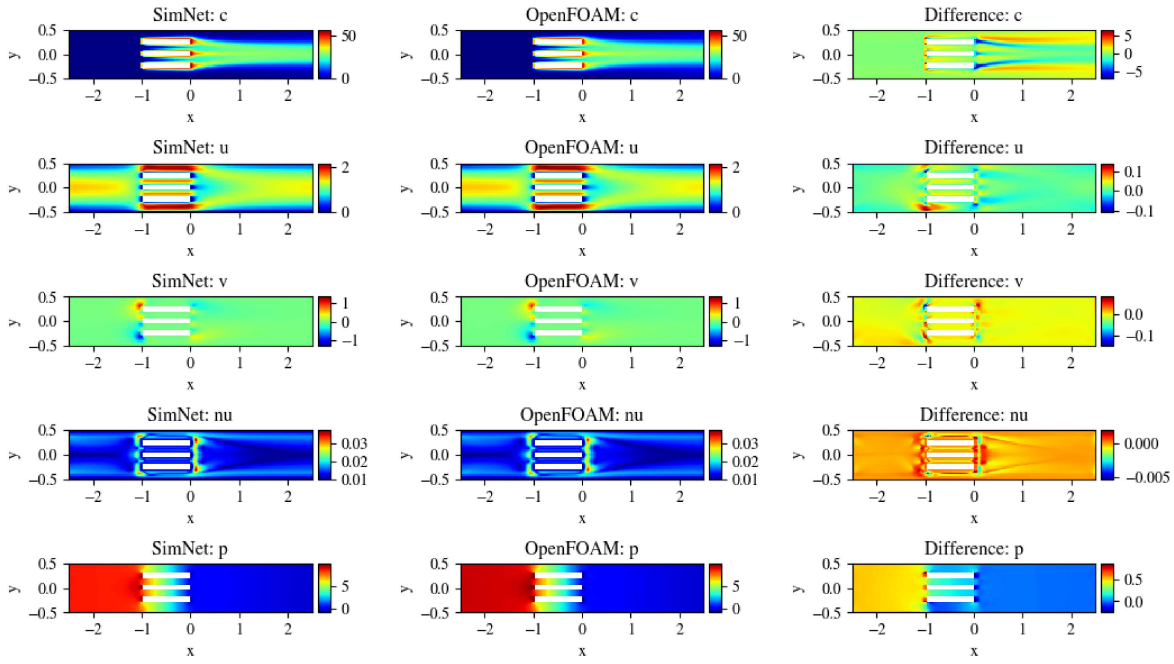


Figure 38: Left: SimNet. Center: OpenFOAM. Right: Difference