14 Forward simulation using STL geometry: Blood Flow in Intracranial Aneurysm

14.1 Introduction

In this tutorial, we will import an STL file for a complicated geometry and use SimNet's SDF library to sample points on the surface and the interior of the STL and train the PINNs to predict flow in this complex geometry. In this tutorial you will learn the following:

1. How to import an STL file in SimNet and sample points in the interior and on the surface of the geometry.

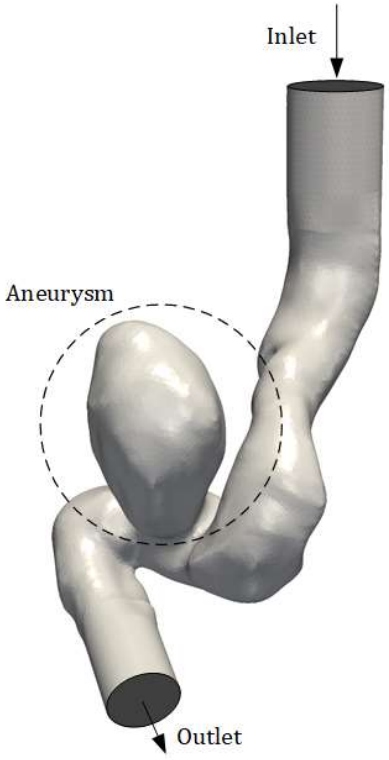


Figure 65: Aneurysm STL file

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of the SimNet user interface. Also, we recommend you to refer to tutorial 3 for information on creating monitor and inference domains. Additionally, to use the modules described in this tutorial, make sure your system satisfies the requirements for SDF library mentioned in Table 1.

Note: For the interior sampling to work, ensure that the STL geometry is watertight. This requirement is not necessary for sampling points on the surface.

14.2 Problem Description

For this simulation we apply a no-slip boundary condition on the walls of the aneurysm $u, v, w = 0$. For the inlet we use a parabolic flow where the flow goes in the normal direction of the inlet and has peak velocity 1.5. The outlet has a zero pressure condition, $p = 0$. The kinematic viscosity of the fluid is 0.025 and the density is a constant 1.0.

14.3 Case Setup

In this tutorial, we will use SimNet's `Mesh` module to sample points using a STL geometry. The module works similar to SimNet's geometry module with methods like `interior_bc`, `boundary_bc` to sample points in the interior and the boundary of the geometry. Currently, we would require separate STL files for each boundary of the geometry and another watertight geometry for sampling points in the interior of the geometry.

Note: All the python scripts for this problem can be found at `examples/aneurysm/`.

Importing the required packages

The list of required packages can be found below. We will import SimNet `Mesh` module to sample points on the STL geometry.

```
import copy
from sympy import Symbol, sqrt, Max
import numpy as np
import tensorflow as tf

from simnet.solver import Solver
from simnet.dataset import TrainDomain, ValidationDomain, InferenceDomain, MonitorDomain
from simnet.data import BC, Validation, Inference, Monitor
from simnet.mesh_utils.mesh import Mesh
from simnet.PDES.navier_stokes import IntegralContinuity, NavierStokes
from simnet.controller import SimNetController
from simnet.csv_utils.csv_rw import csv_to_dict
```

Listing 117: Importing the required packages and modules

14.3.1 Using STL files to create Train domain

The STL geometries can be imported using the `Mesh.from_stl()` function. This function takes in the path of the STL geometry as input. We will need to specify the value of attribute `airtight` as `False` for the open surfaces (eg. boundary STL files).

Then these mesh objects can be sampled either on surface or interior similar to tutorial 2 using the `boundary_bc` or `interior_bc` functions. **Note:** For this tutorial, we will normalize the geometry by scaling it and centering it about the origin (0, 0, 0). This will help in speeding up the training process. Also, we will define a custom function for generating parabolic inlet for the blood vessel.

The code to sample using STL geometry, define all these functions, boundary conditions and generate the `TrainDomain` is shown below.

```
# read stl files to make meshes
point_path = './stl_files/'
inlet_mesh = Mesh.from_stl(point_path + 'aneurysm_inlet.stl', airtight=False)
outlet_mesh = Mesh.from_stl(point_path + 'aneurysm_outlet.stl', airtight=False)
noslip_mesh = Mesh.from_stl(point_path + 'aneurysm_noslip.stl', airtight=False)
integral_mesh = Mesh.from_stl(point_path + 'aneurysm_integral.stl', airtight=False)
interior_mesh = Mesh.from_stl(point_path + 'aneurysm_closed.stl')

# params
nu = 0.025
inlet_vel = 1.5

# inlet velocity profile
def circular_parabola(x, y, z, center, normal, radius, max_vel):
  centered_x = x-center[0]
  centered_y = y-center[1]
  centered_z = z-center[2]
  distance = sqrt(centered_x**2 + centered_y**2 + centered_z**2)
  parabola = max_vel*Max((1 - (distance/radius)**2), 0)
  return normal[0]*parabola, normal[1]*parabola, normal[2]*parabola

# normalize meshes
def normalize_mesh(mesh, center, scale):
  mesh.translate([-c for c in center])
  mesh.scale(scale)

# normalize invars
def normalize_invar(invar, center, scale, dims=2):
  invar['x'] -= center[0]
  invar['y'] -= center[1]
```

```
  invar['z'] -= center[2]
  invar['x'] *= scale
  invar['y'] *= scale
  invar['z'] *= scale
  if 'area' in invar.keys():
    invar['area'] *= scale**dims
  return invar

# scale and normalize mesh and openfoam data
center = (-18.40381048596882, -50.285383353981196, 12.848136936899031)
scale = 0.4
normalize_mesh(inlet_mesh, center, scale)
normalize_mesh(outlet_mesh, center, scale)
normalize_mesh(noslip_mesh, center, scale)
normalize_mesh(integral_mesh, center, scale)
normalize_mesh(interior_mesh, center, scale)
openfoam_invar = normalize_invar(openfoam_invar, center, scale, dims=3)

# geom params
inlet_normal = (0.8526, -0.428, 0.299)
inlet_area = 21.1284*(scale**2)
inlet_center = (-4.24298030045776, 4.082857101816247, -4.637790193399717)
inlet_radius = np.sqrt(inlet_area/np.pi)
outlet_normal = (0.33179, 0.43424, 0.83747)
outlet_area = 12.0773*(scale**2)
outlet_radius = np.sqrt(outlet_area/np.pi)

# define domain
class AneurysmTrain(TrainDomain):
  def __init__(self, **config):
    super(AneurysmTrain, self).__init__()
    # Inlet
    u, v, w = circular_parabola(Symbol('x'),
                                Symbol('y'),
                                Symbol('z'),
                                center=inlet_center,
                                normal=inlet_normal,
                                radius=inlet_radius,
                                max_vel=inlet_vel)
    inlet = inlet_mesh.boundary_bc(outvar_sympy={'u': u, 'v': v, 'w': w},
                                   batch_size_per_area=256)
    self.add(inlet, name="Inlet")

    # Outlet
    outlet = outlet_mesh.boundary_bc(outvar_sympy={'p': 0},
                                     batch_size_per_area=256)
    self.add(outlet, name="Outlet")

    # Noslip
    noslip = noslip_mesh.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                                     batch_size_per_area=32)
    self.add(noslip, name="Noslip")

    # Interior
    interior = interior_mesh.interior_bc(outvar_sympy={'continuity': 0,
                                                       'momentum_x': 0,
                                                       'momentum_y': 0,
                                                       'momentum_z': 0},
                                         batch_size_per_area=128,
                                         batch_per_epoch=1000)
    self.add(interior, name="Interior")

    # Integral Continuity 1
    ic_1 = outlet_mesh.boundary_bc(outvar_sympy={'integral_continuity': 2.540},
                                   lambda_sympy={'lambda_integral_continuity': 0.1},
                                   batch_size_per_area=128)
    self.add(ic_1, name="IntegralContinuity_1")

    # Integral Continuity 2
    ic_2 = integral_mesh.boundary_bc(outvar_sympy={'integral_continuity': -2.540},
                                     lambda_sympy={'lambda_integral_continuity': 0.1},
                                     batch_size_per_area=128)
    self.add(ic_2, name="IntegralContinuity_2")
```

Listing 118: Creating training data from STL files

14.3.2 Creating Validation and Monitor domains

The process of creating `ValidationDomain` and `MonitorDomain` is similar to previous tutorials and hence we would not cover them in detail. We will use the simulation from OpenFOAM for validating our SimNet results. Also, we will create a monitor for pressure drop across the aneurysm to monitor the convergence and compare against OpenFOAM data. The code to generate the these domains can be found below.

```
# read validation data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z', 'U:0': 'u', 'U:1': 'v', 'U:2': 'w', 'p': 'p'}
openfoam_var = csv_to_dict('openfoam/aneurysm_parabolicInlet_sol0.csv', mapping)
openfoam_invar = {key: value for key, value in openfoam_var.items() if key in ['x', 'y', 'z']}
openfoam_outvar = {key: value for key, value in openfoam_var.items() if key in ['u', 'v', 'w', 'p']}

class AneurysmVal(ValidationDomain):
  def __init__(self, **config):
    super(AneurysmVal, self).__init__()
    val = Validation.from_numpy(openfoam_invar, openfoam_outvar)
    self.add(val, name='Val')

class AneurysmMonitor(MonitorDomain):
  def __init__(self, **config):
    super(AneurysmMonitor, self).__init__()
    # metric for pressure drop
    metric = Monitor(inlet_mesh.sample_boundary(16),
                     {'pressure_drop': lambda var: tf.reduce_mean(var['p'])})
    self.add(metric, 'PressureDrop')
```

Listing 119: Defining Validation, Monitor and Inference domains

14.3.3 Making the Neural Network solver

This process is similar to other tutorials. Since we would be solving only laminar flow in this problem, we will use only `NavierStokes` and `IntegralContinuity` equations and define a network similar to tutorial 2. The code to generate the Neural Network solver is shown below.

```
class AneurysmSolver(Solver):
  train_domain = AneurysmTrain
  val_domain = AneurysmVal
  monitor_domain = AneurysmMonitor

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

    self.equations = (NavierStokes(nu=nu*scale, rho=1, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_aneurysm',
        'rec_results_cpu': True,
        'max_steps': 1500000,
        'decay_steps': 15000,
        })

if __name__ == '__main__':
  ctr = SimNetController(AneurysmSolver)
  ctr.run()
```

Listing 120: Defining the Neural Network Solver

14.4 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 aneurysm.py
```

14.5 Results and Post-processing

This tutorial is a good example of over-fitting the training data in the PINNs. Figure 66 shows the comparison of the validation error plots achieved for two different point densities. The case using 10 M points (total points in the .csv file) shows an initial convergence which later diverges even when the training error keeps reducing. This implies that the network is over-fitting the sampled points while sacrificing the accuracy of flow in between them. Increasing the points to 20 M solves that problem and we are able to generalize the flow field to a better resolution.

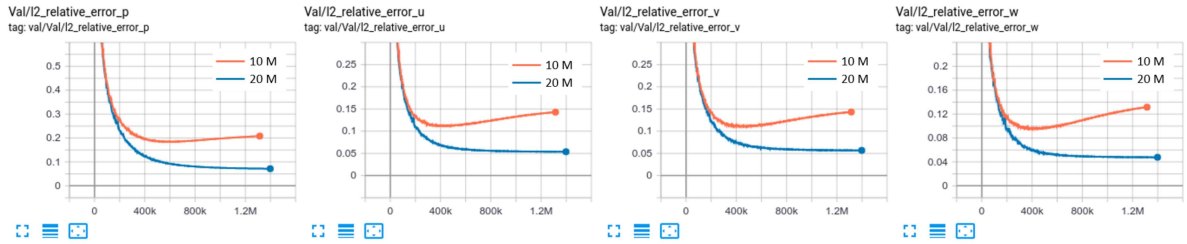


Figure 66: Convergence plots for different point density

Figure 67 shows the pressure developed inside the aneurysm and the vein. A cross-sectional view in figure 68 shows the distribution of velocity magnitude inside the aneurysm. One of the key challenges of this problem is getting the flow to develop inside the aneurysm sac and the streamline plot in figure 69 shows that SimNet successfully captures the flow field inside.

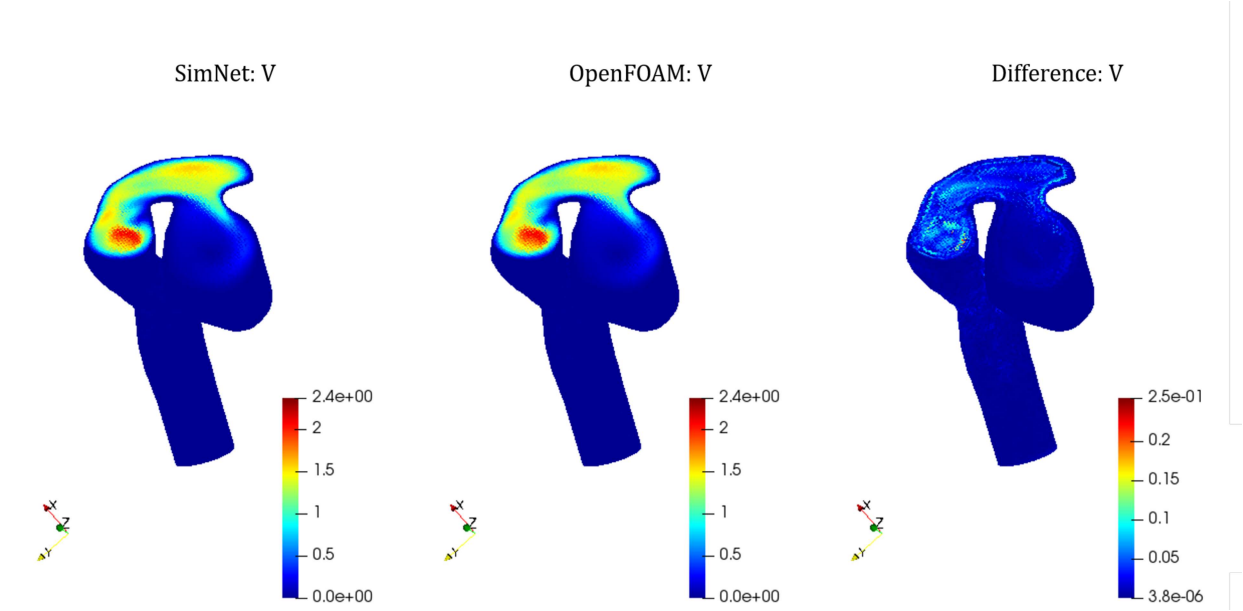


Figure 67: Cross-sectional view aneurysm showing velocity magnitude. Left: SimNet. Center: OpenFOAM. Right: Difference

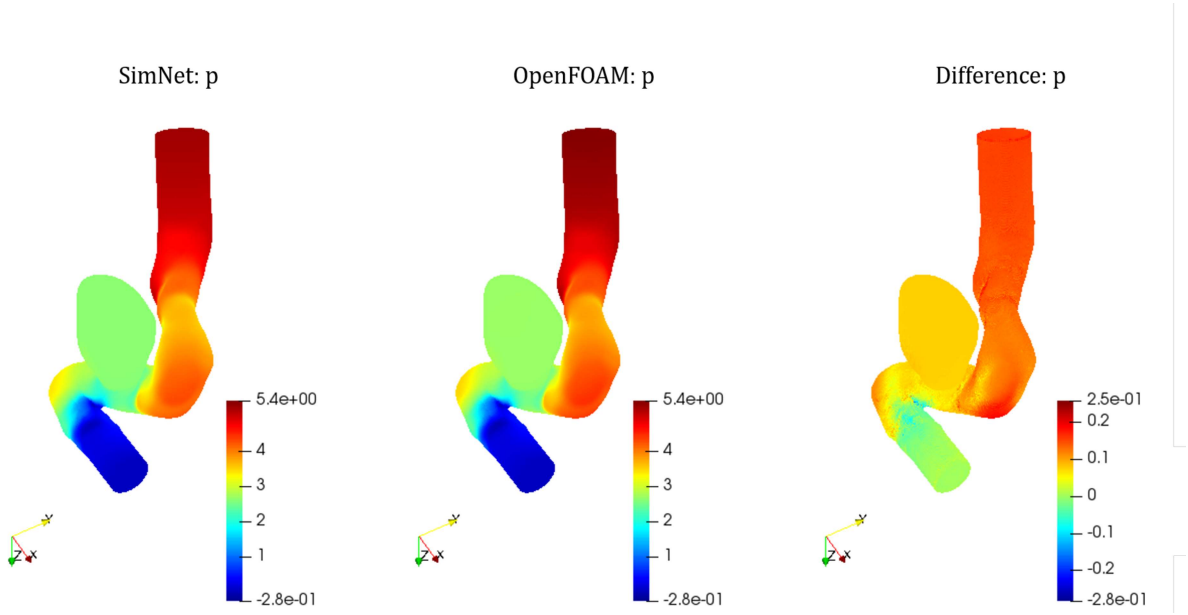


Figure 68: Pressure across aneurysm. Left: SimNet. Center: OpenFOAM. Right: Difference

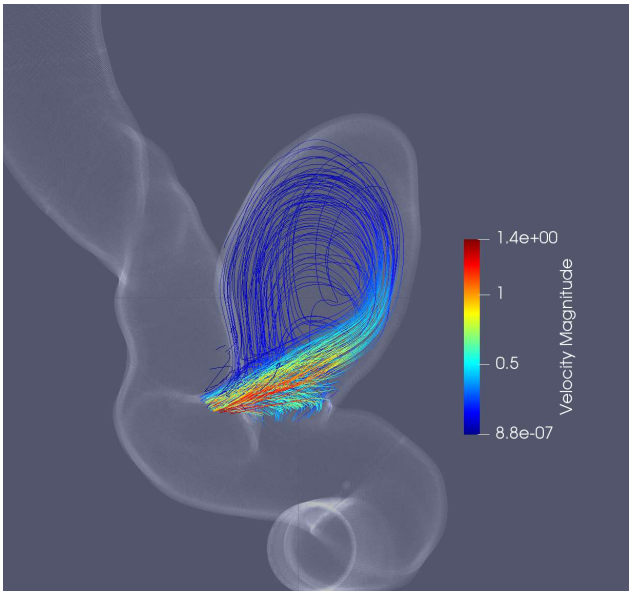


Figure 69: Flow streamlines inside the aneurysm generated from SimNet simulation.

14.6 Accelerating the Training of Neural Network Solvers via Transfer Learning

Numerous applications in science and engineering require repetitive simulations, such as simulation of blood flow in different patient-specific models. Traditional solvers simulate these models independently and from scratch. Even a minor change to the model geometry (such as an adjustment to the patient-specific medical image segmentation) requires a new simulation. Interestingly, and unlike the traditional solvers, neural network solvers can transfer knowledge across different neural network models via transfer learning. In transfer learning, the knowledge acquired by a (source) trained neural network model for a physical system is transferred to another (target) neural network model that is to be trained for a similar physical system with slightly different characteristics (such as geometrical differences). The network parameters of the target model are initialized from the source model, and are re-trained to cope with the new system characteristics without having the neural network model trained from scratch. This transfer of knowledge effectively reduces the time to convergence for neural network solvers. As an example, Figure 70 shows the application of transfer learning in training of neural network solvers for two intracranial aneurysm models with different sac shapes.

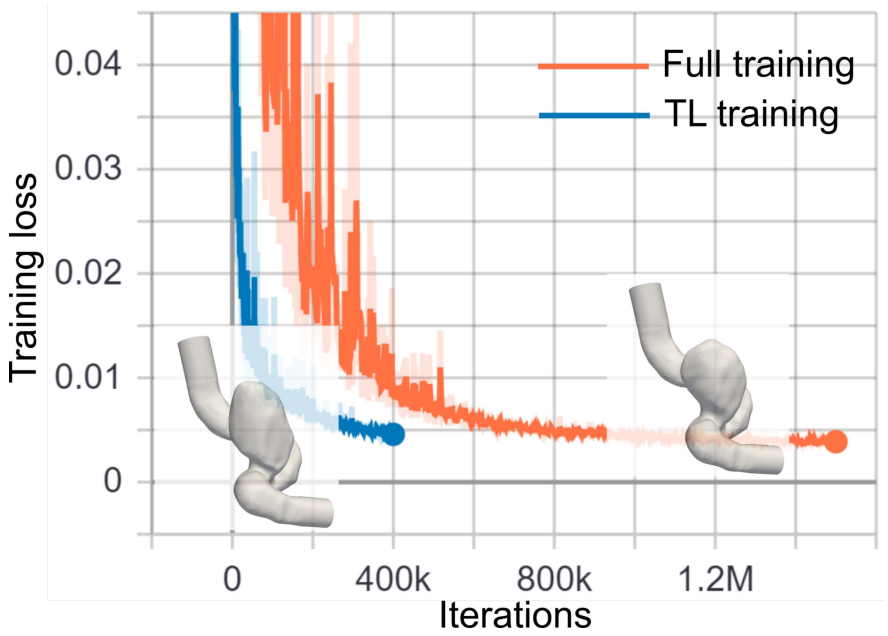


Figure 70: Transfer learning accelerates intracranial aneurysm simulations. Results are for two intracranial aneurysms with different sac shapes.

To use transfer learning in SimNet, we set `'initialize_network_dir'` to the source model network checkpoint. Also, since in transfer learning we fine-tune the source model instead of training from scratch, we use a relatively smaller learning rate compared to a full run, with smaller number of iterations and faster decay, as shown below.

```
@classmethod
def update_defaults(cls, defaults):
  defaults.update({
      'initialize_network_dir': '../aneurysm/network_checkpoint_aneurysm_source',
      'network_dir': './network_checkpoint_aneurysm_target',
      #'max_steps': 1500000,  # full run
      'max_steps': 400000,    # TL run
      #'decay_steps': 15000,  # full run
      'decay_steps': 6000,    # TL run
      #'start_lr': 1e-3,      # full run
      'start_lr': 5e-4,       # TL run
      })
```

Listing 121: Updating the training defaults for Transfer learning

15 Inverse problem: Finding unknown coefficients of a PDE

15.1 Introduction

In this tutorial, we will use SimNet to solve an inverse problem by assimilating data from observations. We will use the flow field computed by OpenFOAM as an input to the PINNs whose job would be to predict the parameters characterizing the flow (eg. viscosity (ν) and thermal diffusivity (α)). In this tutorial you would learn the following:

1. How assimilate analytical/experimental/simulation data in SimNet.
2. How to use the `BC` class in SimNet to create boundary conditions from .csv files.
3. How to use the assimilated data to make predictions of unknown quantities.

Prerequisites

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

15.2 Problem Description

In this tutorial, we will predict the fluid's viscosity and thermal diffusivity by providing the flow field information obtained from OpenFOAM simulations as an input to the PINNs. We will use the same 2D slice from a 3-fin flow field that was used as a validation data in tutorial 8.

To summarize, the training data for this problem would be (u_i, v_i, p_i, T_i) from OpenFOAM simulation and the model would be trained to predict (ν, α) with the constraints of satisfying the governing equations of continuity, Navier-Stokes and advection-diffusion.

The ν and α used for the OpenFOAM simulation are 0.01 and 0.002 respectively.

We will scale T to define a new transport variable c for the advection-diffusion equation as shown in equation 115.

$$\begin{aligned} c &= \frac{T_{actual}}{T_{base}} - 1.0 \\ T_{base} &= 293.498K \end{aligned} \tag{115}$$

As the majority of diffusion of temperature occurs in the wake of the heat sink (Figure 71), we will only sample points in the wake for training the PINNs. We will also discard the points close to the boundary as we will be training the network to minimize loss from the conservation laws alone.

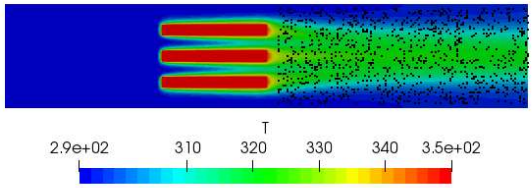


Figure 71: Batch of training points sampled from OpenFOAM data

15.3 Case Setup

In this tutorial we will use `BC` class for making the training data from .csv file. We will make three networks. First network will memorize the flow field by developing a mapping between (x, y) and (u, v, p). Second network will memorize the temperature field by developing a mapping between (x, y) and (c). A third network will be trained to invert out the desired quantities viz. (ν, α). For this problem, we will be using `NavierStokes` and `AdvectionDiffusion` equations from the PDES module.

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

Importing the required packages

The list of packages/modules to be imported are shown below.

```
from sympy import Symbol
import numpy as np
import tensorflow as tf

from simnet.solver import Solver
from simnet.dataset import TrainDomain, InferenceDomain, MonitorDomain
from simnet.data import BC, Monitor, Inference
from simnet.sympy_utils.geometry_2d import Rectangle, Channel2D
from simnet.csv_utils.csv_rw import csv_to_dict
from simnet.PDES.navier_stokes import NavierStokes
from simnet.PDES.advection_diffusion import AdvectionDiffusion
from simnet.controller import SimNetController
```

Listing 122: Importing the required packages and modules

15.3.1 Assimilating data from CSV files/point clouds to create Training data

`BC`

For this problem, we will not be solving the problem for the full domain. Hence, we will not be using the geometry module to create geometry for sampling points. Instead, we will use the point cloud data in form of a .csv file. We will use the `BC` class to handle such input data. The `BC` class takes in separate dictionaries for input variables and output variables. These dictionaries have a key for each variable and a numpy array of values associated to the key. Also, we will provide a batch size for sampling this .csv point cloud. This will be done by specifying the required batch size to the `batch_size` argument.

Since part of the problem involves memorizing the given flow field, we will have `['x', 'y']` as input keys and `['u', 'v', 'p', 'c', 'continuity', 'momentum_x', 'momentum_y', 'advection_diffusion']` as the output keys. Setting `['u', 'v', 'p', 'c']` as input values from OpenFOAM data, we are essentially making the network assimilate the OpenFOAM distribution of these variables in the selected domain. Setting `['continuity', 'momentum_x', 'momentum_y', 'advection_diffusion']` equal to 0, we also inform the network to satisfy the PDE losses at those sampled points. Now, except the ν and α, all the variables in these PDEs are known. Thus the network can use this information to invert out the unknowns.

The code to generate such a boundary condition is shown below.

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

# define geometry
channel = Channel2D((channel_length[0], channel_width[0]),
                    (channel_length[1], channel_width[1]))
heat_sink = Rectangle(heat_sink_origin,
                      (heat_sink_origin[0]+heat_sink_length, heat_sink_origin[1]+heat_sink_fin_thickness))
for i in range(1, nr_heat_sink_fins):
  heat_sink_origin = (heat_sink_origin[0], heat_sink_origin[1]+gap)
  heat_sink = heat_sink + Rectangle(heat_sink_origin,
                                    (heat_sink_origin[0]+heat_sink_length, heat_sink_origin[1]+
    heat_sink_fin_thickness))
geo = channel - heat_sink

# OpenFOAM data
mapping = {'Points:0': 'x', 'Points:1': 'y',
           'U:0': 'u', 'U:1': 'v', 'p': 'p', 'T': 'c'}
openfoam_var = csv_to_dict('openfoam/heat_sink_Pr5_clipped2.csv', mapping)
openfoam_var['c'] = openfoam_var['c']/base_temp - 1.0
openfoam_invar_numpy = {key: value for key, value in openfoam_var.items()
                        if key in ['x', 'y']}
openfoam_outvar_numpy = {key: value for key, value in openfoam_var.items()
                         if key in ['u', 'v', 'p', 'c']}
openfoam_outvar_numpy['continuity'] = np.zeros_like(openfoam_outvar_numpy['u'])
openfoam_outvar_numpy['momentum_x'] = np.zeros_like(openfoam_outvar_numpy['u'])
openfoam_outvar_numpy['momentum_y'] = np.zeros_like(openfoam_outvar_numpy['u'])
```

```
openfoam_outvar_numpy['advection_diffusion']=np.zeros_like(openfoam_outvar_numpy['u'])

class HeatSinkTrain(TrainDomain):
  def __init__(self, **config):
    super(HeatSinkTrain, self).__init__()
    # interior
    interior=BC.from_numpy(openfoam_invar_numpy,openfoam_outvar_numpy,batch_size=1024)
    self.add(interior, name="Interior")
```

Listing 123: Creating training data from .csv files

Note: We have created the geometry using the geometry module only for the sake of Inference domain. The geometry generation part of the code can be omitted if an inference is not required over the entire region.

15.3.2 Creating Monitor and Inference domain

For this tutorial, we will create `MonitorDomain` to monitor the convergence of average `'nu'` and `'D'` inside the domain as the solution progresses. Once we find that the average value of these quantities have reached a steady value, we can end the simulation. The code to generate the `MonitorDomain` can be found below.

```
class HeatSinkMonitor(MonitorDomain):
  def __init__(self, **config):
    super(HeatSinkMonitor, self).__init__()
    global_monitor = Monitor(openfoam_invar_numpy, {'average_D': lambda var: tf.reduce_mean(var['D']),
                                                    'average_nu': lambda var: tf.reduce_mean(var['nu'])})
    self.add(global_monitor, 'GlobalMonitor')

class HeatSinkInference(InferenceDomain):
  def __init__(self, **config):
    super(HeatSinkInference, self).__init__()
    # save entire domain
    x, y = Symbol('x'), Symbol('y')
    interior = Inference(geo.sample_interior(1000, bounds={x: (channel_length), y: (channel_width)}),
                         ['u', 'v', 'p', 'nu', 'c', 'D'])
    self.add(interior, name="Inference")
```

Listing 124: Creating monitor domains

Note: It is also possible to create `InferenceDomain` to visualize the distribution of `'nu'` and `'D'` inside the flow field.

15.3.3 Making the Neural Network Solver for a Inverse problem

The process of creating a neural network for an inverse problem is similar to most of the problems we have seen in previous tutorials. However, as the information for the flow variables, and in turn their gradients, is already present (from OpenFOAM data) for the network to memorize, we will stop the gradient calls on each of these variables in their respective equations. This means that only the networks predicting `'nu'` and `'D'` will be optimized to minimize the equation residuals. The velocity, pressure and their gradients are treated as ground truth data.

Also, note that the the viscosity and diffusivity are passed in as Symbolic variables (`'nu'` and `'D'` respectively) to the equations as they are unknowns in this problem.

Similar to tutorial 8, we will train two separate networks to memorize flow variables (u, v, p), and scalar transport variable (c). Also, because `'nu'` and `'D'` are the custom variables we defined, we will have a separate network `'invert_net'` that produces ν and α at the output nodes.

The code to generate the neural network can be found below.

```
nu = Symbol('nu')
D = Symbol('D')

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

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

    self.equations = (NavierStokes(nu=nu, rho=1.0, dim=2, time=False).make_node(stop_gradients=['u', 'u__x', '
     u__x__x', 'u__y', 'u__y__y',
```

```
                                                                                                    'v', 'v__x', '
    v__x__x', 'v__y', 'v__y__y',
                                                                                                    'p', 'p__x', '
    p__y'])
                    + AdvectionDiffusion(T='c', rho=1.0, D=D, dim=2, time=False).make_node(stop_gradients=['u'
    ,
                                                                                                                'v'
    ,
                                                                                                                'c'
    , 'c__x', 'c__x__x', 'c__y', 'c__y__y']))
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    heat_net = self.arch.make_node(name='heat_net',
                                   inputs=['x', 'y'],
                                   outputs=['c'])
    invert_net = self.arch.make_node(name='invert_net',
                                     inputs=['x', 'y'],
                                     outputs=['nu', 'D'])

    self.nets = [flow_net, heat_net, invert_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_heat_sink_inverse',
        'start_lr': 5e-4,
        'max_steps': 100000,
        'decay_steps': 1000,
        })

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

Listing 125: Making the SimNet solver for an Inverse problem

15.4 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_inverse.py
```

15.5 Results and Post-processing

We can monitor the Tensorboard plots to see the convergence of the simulation. The Tensorboard graphs should look similar to the ones shown in figure 72.

Table 5: Comparison of the inverted coefficients with the actual values

Property	OpenFOAM (True)	SimNet (Predicted)
Kinematic Viscosity (m^2/s)	1.00×10^{-2}	1.03×10^{-2}
Thermal Diffusivity (m^2/s)	2.00×10^{-3}	2.19×10^{-3}

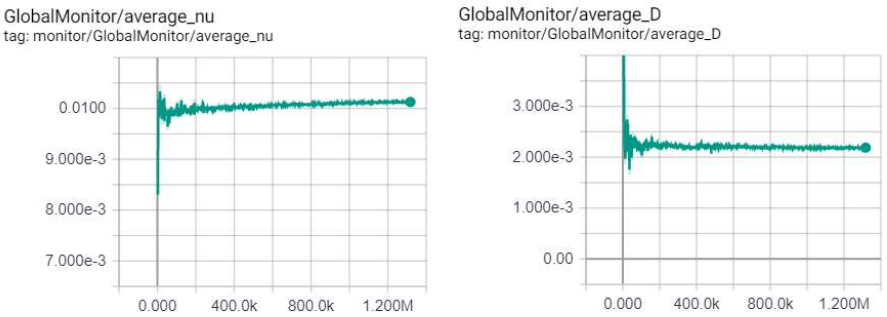


Figure 72: Tensorboard plots for ν and α

16 Parameterized Simulations and Design Optimization: 3D heat sink

16.1 Introduction

In this tutorial, we will walk through the process of simulating a parameterized problem using SimNet. The neural networks in SimNet allow us to solve problems for multiple design parameters in a single training. This implies that once the training is complete, it is possible to run inference on several geometry/physical parameter combinations as a post-processing step, without solving the forward problem again. We will see that such parameterization increases the computational cost only fractionally while solving the entire desired design space.

To demonstrate this feature we will solve the flow and heat over a 3-fin heat sink whose fin height, fin thickness, and fin length are variable. We will then perform a design optimization to find out the most optimal fin configuration for our heat sink example. By the end of this tutorial, you would learn to easily convert any simulation to a parametric study using SimNet's Geometry module and Neural Network solver, and also perform design optimization. In this tutorial, you would learn the following:

1. How to set up a parametric simulation in SimNet.
2. How to perform design optimization.

Prerequisites

This tutorial is an extension of tutorial 13 where we discussed how to use SimNet for solving Conjugate Heat problems. In this tutorial, we take up the same geometry setup and solve it for a parameterized setup at an increased Reynolds number. Hence, we recommend you to refer tutorial 13 for any additional details related to geometry specification and boundary conditions.

Note: In this tutorial we will focus on parameterization which is independent of the physics being solved and can be applied to any class of problems covered in the User Guide.

16.2 Problem Description

Please refer the geometry and boundary conditions for a 3-fin heat sink in tutorial 13. We will parameterize this problem to solve for several heat sink designs in a single neural network training. We will modify the heat sink's fin dimensions (thickness, length and height) to create a design space of various heat sinks. In this tutorial, we change the Reynolds number of 50 in tutorial 13 and increase it to 500 and incorporate turbulence using Zero Equation turbulence model.

More accurately, for this problem, we will vary the height (h), length (l), and thickness (t) of the central fin and the two side fins. The height, length, and thickness of the two side fins are kept the same, and therefore, there will be a total of six geometry parameters. The ranges of variation for these geometry parameters are given in equation 116.

$$\begin{aligned} h_{centralfin} &= (0.0, 0.6), \\ h_{sidefins} &= (0.0, 0.6), \\ l_{centralfin} &= (0.5, 1.0) \\ l_{sidefins} &= (0.5, 1.0) \\ t_{centralfin} &= (0.05, 0.15) \\ t_{sidefins} &= (0.05, 0.15) \end{aligned} \tag{116}$$

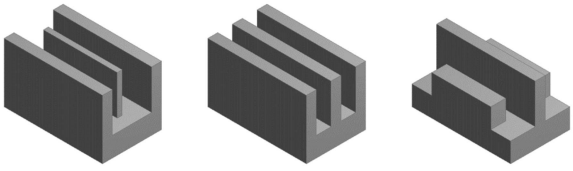


Figure 73: Examples of some of the 3 Fin geometries covered in the chosen design space

16.3 Case Setup

In this tutorial, we will use the 3D geometry module from SimNet to create the parameterized 3-fin heat sink geometry. The current version of SimNet does not support the parameterization of some of the discrete features like the number of fins. Hence in this tutorial we will only cover parameters that are continuous. Also, we will be training the parameterized model and validating it by performing inference on a case where $h_{centralfin} = 0.4$, $h_{sidefins} = 0.4$, $l_{centralfin} = 1.0$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.1$, and $t_{sidefins} = 0.1$. At the end of the tutorial we will also present the comparison between results for the above combination of parameters obtained from a parameterized model versus results obtained from a non-parameterized model trained on just a single geometry corresponding to the same set of values. This will highlight the usefulness of using PINNs for doing parameterized simulations in comparison to some of the traditional methods.

As we did in the tutorial 13, we will split the code in three parts, the domain, flow solver and heat solver. The design optimization would be run as a post-processing step after both the flow and heat are trained using a separate script.

Hence, we will split the code into the following different parts:

1. `three_fin_domain_zero_eq_parameterized.py`: This file will contain all the definitions of geometry, boundary conditions, equations, and definition of validation, monitor, and inference domains.
2. `three_fin_flow_solver_zero_eq_parameterized.py`: This file will contain the solver details for flow. It will derive all the required information from the `three_fin_domain_zero_eq_parameterized.py` file which would be imported at the beginning of the code.
3. `three_fin_heat_solver_zero_eq_parameterized.py`: This file will contain the solver details for heat. It will derive all the required information from the `three_fin_domain_zero_eq_parameterized.py` file which would be imported at the beginning of the code.
4. `three_fin_optimizer_zero_eq_parameterized.py`: This file will contain the brute force optimization details and it uses the trained checkpoints generated by the solver files above.

Note: All the files for this problem can be found at `examples/three_fin_3d/turbulent_parameterized_with_design_optimization/`.

Importing the required packages

The list of packages/modules to be imported in each of the file are similar to tutorial 13. So we will skip the description here. Only additional packages that we would import for this tutorial are `ZeroEquation` from `PDES` module for turbulence. The detailed packages imported can be found in the scripts discussed in the previous section.

16.3.1 Creating the Parameterized Geometry

As described earlier, we will edit the `three_fin_domain_zero_eq_parameterized.py` file to create the geometry. We will use all the same primitives that we used in the tutorial 13.

To create the parameteric shapes and solve them for any problem in SimNet, there are 4 things that we need to in addition to a regular single geometry problem.

1. Create symbolic variables for the dimensions we want to parameterize
2. Define ranges of variation for all the variables created in step 1
3. Generate geometry using SimNet's geometry module by supplying the symbolic variables as inputs
4. Pass the parameteric ranges to the `param_ranges` attribute of `interior_bc` and `boundary_bc`.

Then, any symbolic parameters that we may have created (geometric or non geometric) need to be passed as inputs in addition to the Cartesian coordinates while making the neural network in the solver file.

Hence, for this problem, we pass in the symbolic variables for the geometric dimensions. We will create symbolic variables for length, thickness and height for the side fins and center/middle fins separately using sympy `Symbol`.

The code to generate the parameterized geometry can be found below (Note: In the scripts, the fin height for is measured from the top of the heat sink base):

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

```
# parametric variation
fin_height_m, fin_height_s = Symbol('fin_height_m'), Symbol('fin_height_s')
fin_length_m, fin_length_s = Symbol('fin_length_m'), Symbol('fin_length_s')
fin_thickness_m, fin_thickness_s = Symbol('fin_thickness_m'), Symbol('fin_thickness_s')
height_m_range = (0.0, 0.6)
height_s_range = (0.0, 0.6)
length_m_range = (0.5, 1.0)
length_s_range = (0.5, 1.0)
thickness_m_range = (0.05, 0.15)
thickness_s_range = (0.05, 0.15)
param_ranges = {fin_height_m: height_m_range,
                fin_height_s: height_s_range,
                fin_length_m: length_m_range,
                fin_length_s: length_s_range,
                fin_thickness_m: thickness_m_range,
                fin_thickness_s: thickness_s_range}

# geometry params for domain
channel_origin        = (-2.5, -0.5, -0.5)
channel_dim           = (5.0, 1.0, 1.0)
heat_sink_base_origin = (-1.0, -0.5, -0.3)
heat_sink_base_dim    = (1.0, 0.2, 0.6)
fin_origin            = (heat_sink_base_origin[0]+0.5-fin_length_s/2, -0.3, -0.3)
fin_dim               = (fin_length_s, fin_height_s, fin_thickness_s)       # two side fins
total_fins            = 2                                                   # two side fins
flow_box_origin       = (-1.1, -0.5, -0.5)
flow_box_dim          = (1.6, 1.0, 1.0)
source_origin         = (-0.7, -0.5, -0.1)
source_dim            = (0.4, 0.0, 0.2)
source_area           = 0.08

# params for simulation
# fluid params
nu                    = 0.002
rho                   = 1
inlet_vel             = 1.0
volumetric_flow       = 1.0
# heat params
D_solid               = 0.0625
D_fluid               = 0.02
inlet_t               = 293.15
grad_t                = 360
inlet_t = inlet_t/273.15 - 1.0
grad_t = grad_t/273.15

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

# define geometry
# channel
channel = Channel(channel_origin, (channel_origin[0]+channel_dim[0],
                                   channel_origin[1]+channel_dim[1],
                                   channel_origin[2]+channel_dim[2]))

# three fin heat sink
heat_sink_base = Box(heat_sink_base_origin, (heat_sink_base_origin[0]+heat_sink_base_dim[0], # base of heat
    sink
                                             heat_sink_base_origin[1]+heat_sink_base_dim[1],
                                             heat_sink_base_origin[2]+heat_sink_base_dim[2]))
fin_center = (fin_origin[0] + fin_dim[0]/2, fin_origin[1] + fin_dim[1]/2, fin_origin[2] + fin_dim[2]/2)
fin = Box(fin_origin, (fin_origin[0]+fin_dim[0],
                       fin_origin[1]+fin_dim[1],
                       fin_origin[2]+fin_dim[2]))
gap = (heat_sink_base_dim[2]-fin_dim[2])/(total_fins-1) # gap between fins
fin.repeat(gap, repeat_lower=(0,0,0), repeat_higher=(0,0,total_fins-1), center=fin_center)
three_fin = heat_sink_base + fin

# parameterized center fin
center_fin_origin = (heat_sink_base_origin[0]+0.5-fin_length_m/2, fin_origin[1], -fin_thickness_m/2)
center_fin_dim = (fin_length_m, fin_height_m, fin_thickness_m)
center_fin = Box(center_fin_origin, (center_fin_origin[0]+center_fin_dim[0],
                                     center_fin_origin[1]+center_fin_dim[1],
                                     center_fin_origin[2]+center_fin_dim[2]))
three_fin = three_fin + center_fin

# entire geometry
geo = channel - three_fin

# low and high resolution geo away and near the heat sink
flow_box = Box(flow_box_origin, (flow_box_origin[0]+flow_box_dim[0], # base of heat sink
                                 flow_box_origin[1]+flow_box_dim[1],
```

```
                                  flow_box_origin[2]+flow_box_dim[2]))
lr_geo = geo - flow_box
hr_geo = geo & flow_box
lr_bounds_x = (channel_origin[0], channel_origin[0] + channel_dim[0])
lr_bounds_y = (channel_origin[1], channel_origin[1] + channel_dim[1])
lr_bounds_z = (channel_origin[2], channel_origin[2] + channel_dim[2])
hr_bounds_x = (flow_box_origin[0], flow_box_origin[0] + flow_box_dim[0])
hr_bounds_y = (flow_box_origin[1], flow_box_origin[1] + flow_box_dim[1])
hr_bounds_z = (flow_box_origin[2], flow_box_origin[2] + flow_box_dim[2])

# inlet and outlet
inlet = Plane(channel_origin, (channel_origin[0], channel_origin[1]+channel_dim[1], channel_origin[2]+
    channel_dim[2]), -1)
outlet = Plane((channel_origin[0]+channel_dim[0], channel_origin[1], channel_origin[2]), (channel_origin[0]+
    channel_dim[0], channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]), 1)

# planes for integral continuity
x_pos = Symbol('x_pos')
integral_plane = Plane((x_pos, channel_origin[1], channel_origin[2]),
                       (x_pos, channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]),
                       1)
x_pos_range = {x_pos: lambda batch_size: np.full((batch_size, 1), np.random.uniform(-1.1, 0.1))}
fixed_param_range = {fin_height_m:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0, 0.6)),
                     fin_height_s:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0, 0.6)),
                     fin_length_m:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.5, 1.0)),
                     fin_length_s:    lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.5, 1.0)),
                     fin_thickness_m: lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.05, 0.15))
    ,
                     fin_thickness_s: lambda batch_size: np.full((batch_size, 1), np.random.uniform(0.05, 0.15))
    }
```

Listing 126: Generating the parameterized geometry

16.3.2 Defining the Boundary conditions and Equations for a parameterized problem

The boundary conditions for the flow and heat are similar to the tutorial 13, so we will again skip the description of them here. Key thing to note for parameterized problems is that, we need to input the `param_ranges=param_ranges` to each of the boundary conditions (`boundary_bc`, `interior_bc`) created to ensure we are solving the parameterized problem. For the parameterized integral continuity planes, we would combine the fixed parameter dictionary with the x position dictionary of IC planes using `{**x_pos_range, **fixed_param_range}` dictionary.

Note: For simplicity, we just show the flow boundary conditions and equations for a parameterized problem here. Similar steps can be taken to defined the heat boundary conditions. The detailed script can be referred in the `examples/` directory for the exact definition of each boundary condition and equation.

```
# define flow domain
class ThreeFinFlowTrain(TrainDomain):
  def __init__(self, **config):
    super(ThreeFinFlowTrain, self).__init__()
    # inlet
    inletBC = inlet.boundary_bc(outvar_sympy={'u': inlet_vel, 'v': 0, 'w': 0},
                                batch_size_per_area=500,
                                lambda_sympy={'lambda_u': channel.sdf, # weight zero on edges
                                              'lambda_v': 1.0,
                                              'lambda_w': 1.0},
                                criteria=Eq(x, channel_origin[0]),
                                param_ranges=param_ranges)
    self.add(inletBC, name="Inlet")

    # outlet
    outletBC = outlet.boundary_bc(outvar_sympy={'p': 0},
                              batch_size_per_area=500,
                              criteria=Eq(x, channel_origin[0]+channel_dim[0]),
                              param_ranges=param_ranges)
    self.add(outletBC, name="Outlet")

    # no slip for channel walls
    no_slip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                              batch_size_per_area=500,
                              param_ranges=param_ranges)
    self.add(no_slip, name="NoSlipChannel")

    # flow interior low res away from three fin
    interiorF_lr = lr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                      bounds={x: lr_bounds_x, y: lr_bounds_y, z: lr_bounds_z},
```

```
                                    lambda_sympy={'lambda_continuity': geo.sdf,
                                                  'lambda_momentum_x': geo.sdf,
                                                  'lambda_momentum_y': geo.sdf,
                                                  'lambda_momentum_z': geo.sdf},
                                    batch_size_per_area=500,
                                    param_ranges=param_ranges)
    self.add(interiorF_lr, name="FlowInterior_LR")

    # flow interiror high res near three fin
    interiorF_hr = hr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                    bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                    lambda_sympy={'lambda_continuity': geo.sdf,
                                                  'lambda_momentum_x': geo.sdf,
                                                  'lambda_momentum_y': geo.sdf,
                                                  'lambda_momentum_z': geo.sdf},
                                    batch_size_per_area=3000,
                                    param_ranges=param_ranges)
    self.add(interiorF_hr, name="FlowInterior_HR")

    # integral continuity
    for i in range(5):
      IC = integral_plane.boundary_bc(outvar_sympy={'integral_continuity': 1.0},
                                    batch_size_per_area=10000,
                                    lambda_sympy={'lambda_integral_continuity': 1.0},
                                    criteria=geo.sdf>0,
                                    param_ranges={**x_pos_range, **fixed_param_range},
                                    fixed_var=False)
      self.add(IC, name="IntegralContinuity_"+str(i))
```

Listing 127: Defining the flow boundary conditions and equations to solve

16.3.3 Creating Validation, Monitor and Inference domain for a parameterized simulation

We will use the validation data from OpenFOAM that was generated for the following combination of parameters: $h_{centralfin} = 0.4$, $h_{sidefins} = 0.4$, $l_{centralfin} = 1.0$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.1$, and $t_{sidefins} = 0.1$. This will be done by updating the .csv file from the OpenFOAM simulation with the above values for the newly defined parameters. The code to do the same can be found below.

Again, for simplicity, we only show the validation domain for the flow field's flow part. Similar process needs to be followed for the validation domain of the heat part where we include both fluid and solid and the details can be referred in the actual scripts.

```
# validation data
# flow data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z',
           'U:0': 'u', 'U:1': 'v','U:2': 'w', 'p_rgh': 'p', 'T': 'theta_f'}
openfoam_var = csv_to_dict('../openfoam/threeFin_extend_zeroEq_re500_fluid.csv', mapping)
openfoam_var['theta_f'] = openfoam_var['theta_f']/273.15 - 1.0 # normalize heat
openfoam_var['x'] = openfoam_var['x'] + channel_origin[0]
openfoam_var['y'] = openfoam_var['y'] + channel_origin[1]
openfoam_var['z'] = openfoam_var['z'] + channel_origin[2]
openfoam_var.update({'fin_height_m': np.full_like(openfoam_var['x'], 0.4)})
openfoam_var.update({'fin_height_s': np.full_like(openfoam_var['x'], 0.4)})
openfoam_var.update({'fin_length_m': np.full_like(openfoam_var['x'], 1.0)})
openfoam_var.update({'fin_length_s': np.full_like(openfoam_var['x'], 1.0)})
openfoam_var.update({'fin_thickness_m': np.full_like(openfoam_var['x'], 0.1)})
openfoam_var.update({'fin_thickness_s': np.full_like(openfoam_var['x'], 0.1)})
openfoam_invar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['x','y','z','
     fin_height_m','fin_length_m','fin_thickness_m','fin_height_s','fin_length_s','fin_thickness_s']}
openfoam_outvar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['u','v','w','p']}
openfoam_outvar_flow_heat_numpy = {key: value for key, value in openfoam_var.items() if key in ['theta_f']}

class ThreeFinFlowVal(ValidationDomain):
  def __init__(self, **config):
    super(ThreeFinFlowVal, self).__init__()
    # fluid validation
    val = Validation.from_numpy(openfoam_invar_flow_numpy, openfoam_outvar_flow_numpy)
    self.add(val, name='ValFlowField')
```

Listing 128: Defining the validation domain for a parametric simulation

For this problem, for the flow part, we monitored the residuals of the continuity and momentum equations along with pressure drop across the heat sink in tutorial 13. The process to define the `MonitorDomain` domains is similar, but as

an addition, for parametric simulations, we will have to supply the values for the parameters form which we would like to monitor these values. Like validation data, we will monitor these values for the following combination of parameters: $h_{centralfin} = 0.6$, $h_{sidefins} = 0.6$, $l_{centralfin} = 1.0$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.1$, and $t_{sidefins} = 0.1$. The definition of the heat domains can be referred in the scripts.

```
class ThreeFinFlowMonitor(MonitorDomain):
  def __init__(self, **config):
    super(ThreeFinFlowMonitor, self).__init__()
    # metric for equation residuals
    global_monitor = Monitor(geo.sample_interior(10000, bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
      param_ranges={fin_height_m: 0.4, fin_height_s: 0.4, fin_length_m: 1.0, fin_length_s: 1.0, fin_thickness_m:
      0.1, fin_thickness_s: 0.1}),
                            {'mass_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['continuity'])),

                            'momentum_x_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_x'])),
                            'momentum_y_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_y'])),
                            'momentum_z_imbalance': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     momentum_z']))})
    self.add(global_monitor, 'ResiduallMonitor')

    # metric for pressure drop front
    p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges={x_pos: heat_sink_base_origin[0]-
     heat_sink_base_dim[0], fin_height_m: 0.4, fin_height_s: 0.4, fin_length_m: 1.0, fin_length_s: 1.0,
     fin_thickness_m: 0.1, fin_thickness_s: 0.1}),
                        {'pressure': lambda var: tf.reduce_mean(var['p'])})
    self.add(p_monitor, 'FrontPressure')
    p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges={x_pos: heat_sink_base_origin[0]+2*
     heat_sink_base_dim[0], fin_height_m: 0.4, fin_height_s: 0.4, fin_length_m: 1.0, fin_length_s: 1.0,
     fin_thickness_m: 0.1, fin_thickness_s: 0.1}),
                        {'pressure': lambda var: tf.reduce_mean(var['p'])})
    self.add(p_monitor, 'BackPressure')
```

Listing 129: Defining the monitor domain for a parameteric simulation

16.3.4 Making the Neural Network Solver for a parameterized problem

Once all the parameterized domain definitions are completed, for training the parameterized model, we will have the symbolic parameters we defined earlier as inputs to the neural network architecture in `flow_net` viz. `'fin_height_m'`, `'fin_height_s'`, `'fin_thickness_m'`, `'fin_thickness_s'`, `'fin_length_m'`, `'fin_length_s'` along with the usual x, y, z coordinates. The outputs remain the same as what we would have for any other non-parameterized simulation. The below code shows the flow solver file for the paramterized problem.

```
class ThreeFinFlowSolver(Solver):
  train_domain = ThreeFinFlowTrain
  monitor_domain = ThreeFinFlowMonitor
  val_domain = ThreeFinFlowVal

  def __init__(self, **config):
    super(ThreeFinFlowSolver, self).__init__(**config)
    self.equations = (NavierStokes(nu='nu', rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node()
                      + ZeroEquation(nu=nu, dim=3, time=False, max_distance=0.5).make_node()
                      + [Node.from_sympy(geo.sdf, 'normal_distance')])
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z',
                                   'fin_height_m', 'fin_height_s',
                                   'fin_length_m', 'fin_length_s',
                                   'fin_thickness_m', 'fin_thickness_s'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_three_fin_fluid_flow_parameterized_zero_eq',
        'rec_results_cpu': True,
        'max_steps': 1500000,
        'start_lr': 5e-4
        })

if __name__ == '__main__':
  ctr = SimNetController(ThreeFinFlowSolver)
  ctr.run()
```

Listing 130: Making the SimNet solver for parameterized simulations

16.4 Running the SimNet solver

This part is exactly similar to tutorial 13 and once all the definitions are complete, we can execute the parameterized problem like any other problem. The sequence in which the scripts need to be executed can be summarized as follows:

1. Solve the flow equations first (run `three_fin_flow_solver_zero_eq_parameterized.py`)
2. After convergence of flow, solve the heat equations (run `three_fin_heat_solver_zero_eq_parameterized.py`)
3. Optional design optimization: After convergence of heat, run inference for finding the optimal design (run `three_fin_optimizer_zero_eq_parameterized.py`)

16.5 Design Optimization

As we discussed previously, we can optimize the design once the training is complete by making only some minor modifications. A typical design optimization usually contains an objective function that we intend to minimize/maximize subject to some physical/design constraints.

For heat sink designs, usually the peak temperature that can be reached at the source chip is limited. This limit arises from the operating temperature requirements of the chip on which the heat sink is mounted for cooling purposes. The design is then constrained by the maximum pressure drop that can be successfully provided by the cooling system that pushes the flow around the heat sink. Mathematically this can be expressed as below:

Table 6: Optimization problem

	Variable/Function	Description
minimize	$Peak\ Temperature$	Minimize the peak temperature at the source chip
with respect to	$h_{centralfin}$, $h_{sidefins}$, $l_{centralfin}$, $l_{sidefins}$, $t_{centralfin}$, $t_{sidefins}$	Geometric Design Variables of the Heat Sink
subject to	$Pressure\ drop < 2.5$	Limit on the pressure drop. Maximum value of the pressure drop that can be provided by the cooling system

Let's now see how to tackle this problem using SimNet.

As you have noticed, while solving the parameterized simulation we created some monitors to track the peak temperature and the pressure drop for some design variable combination. We basically would follow the same process and use the `Monitor` domain to find the values for multiple combinations of the design variables this time. We can create this simply by looping through the multiple designs. Since we would be looping through several variable combinations, we recommend to add these monitors only after the training is complete to achieve better computational efficiency.

The following snippet of code needs to be added to the domain file (`three_fin_domain_zero_eq_parameterized.py`) to be able to sample the pressure and temperature values for the required designs.

```
class ThreeFinDesignOptMonitor(MonitorDomain):
  def __init__(self, **config):
    super(ThreeFinDesignOptMonitor, self).__init__()

    # define candidate designs
    num_samples = 3
    inference_param_tuple  = itertools.product(np.linspace(*height_m_range, num_samples),
                                               np.linspace(*height_s_range, num_samples),
                                               np.linspace(*length_m_range, num_samples),
                                               np.linspace(*length_s_range, num_samples),
                                               np.linspace(*thickness_m_range, num_samples),
```

```
                                            np.linspace(*thickness_s_range, num_samples))
    for (HS_height_m_, HS_height_s_, HS_length_m_, HS_length_s_, HS_thickness_m_, HS_thickness_s_) in
     inference_param_tuple:
      HS_height_m = float(HS_height_m_)
      HS_height_s = float(HS_height_s_)
      HS_length_m = float(HS_length_m_)
      HS_length_s = float(HS_length_s_)
      HS_thickness_m = float(HS_thickness_m_)
      HS_thickness_s = float(HS_thickness_s_)
      specific_param_ranges = {fin_height_m: HS_height_m,
                               fin_height_s: HS_height_s,
                               fin_length_m: HS_length_m,
                               fin_length_s: HS_length_s,
                               fin_thickness_m: HS_thickness_m,
                               fin_thickness_s: HS_thickness_s}

      # add metrics for pressure drop
      plane_param_ranges = {**specific_param_ranges, **{x_pos: heat_sink_base_origin[0]-heat_sink_base_dim[0]}}
      p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges=plane_param_ranges),
                          {'pressure': lambda var: tf.reduce_mean(var['p'])})
      self.add(p_monitor, 'FP_'+str(HS_height_m)+'_'+str(HS_height_s)+'_'+str(HS_length_m)+'_'
                               +str(HS_length_s)+'_'+str(HS_thickness_m)+'_'+str(HS_thickness_s))

      plane_param_ranges = {**specific_param_ranges, **{x_pos: heat_sink_base_origin[0]+2*heat_sink_base_dim
     [0]}}
      p_monitor = Monitor(integral_plane.sample_boundary(50000, param_ranges=plane_param_ranges),
                          {'pressure': lambda var: tf.reduce_mean(var['p'])})
      self.add(p_monitor, 'BP_'+str(HS_height_m)+'_'+str(HS_height_s)+'_'+str(HS_length_m)+'_'
                               +str(HS_length_s)+'_'+str(HS_thickness_m)+'_'+str(HS_thickness_s))

      # add metrics for peak temp
      plane_param_ranges = {**specific_param_ranges}
      temp_monitor = Monitor(three_fin.sample_boundary(10000, criteria=Eq(y, source_origin[1]), param_ranges=
     plane_param_ranges),
                         {'peak_temp': lambda var: tf.reduce_max(var['theta_s'])})
      self.add(temp_monitor, 'PT_'+str(HS_height_m)+'_'+str(HS_height_s)+'_'+str(HS_length_m)+'_'
                               +str(HS_length_s)+'_'+str(HS_thickness_m)+'_'+str(HS_thickness_s))
```

Listing 131: Making the Monitors for sampling multiple design configurations

Once the monitors are defined in the domain file, all we need to do is run the solver files in the `'eval'` mode to be able to produce inference on all the sampled variable combinations.

This can be achieved using the flag `--run_mode=eval` while executing the solver python scripts again after the initial training. This will populate the `/network_checkpoint_.../monitor_domain/results/` directory with all the designs and their corresponding peak temperature and pressure drops for us to look at. We can then select the design with the least peak temperature and the maximum pressure drop < 2.5.

Doing this process manually is tedious as it requires going through several csv files and then finding the optimal. Alternatively, the same thing can be written in a script. Below we show the contents of `three_fin_optimizer_zero_eq_parameterized.py`. This file basically reads in the new monitor domain we defined, executes the required scripts in `'eval'` mode and then ranks the designs in the order of most optimal to least optimal under the specified constraints.

```
# import domain
from three_fin_domain_zero_eq_parameterized import ThreeFinHeatTrain, ThreeFinDesignOptMonitor
from three_fin_heat_solver_zero_eq_parameterized import ThreeFinHeatSolver

# import SimNet library
from simnet.solver import Solver
from simnet.controller import SimNetController
from simnet.csv_utils.csv_rw import dict_to_csv

# import other libraries
import numpy as np
import os
import csv

# specify the design optimization requirements
max_pressure_drop = 2.5
num_design = 10
path = './network_checkpoint_three_fin_heat_parameterized_zero_eq'
invar_mapping = ['fin_height_middle',
                 'fin_height_sides',
                 'fin_length_middle',
                 'fin_length_sides',
```

```
                'fin_thickness_middle',
                'fin_thickness_sides']
outvar_mapping = ['pressure_drop', 'peak_temp']

# run the monitor domain for front and back pressures and the peak temp
class ThreeFinDesignOpt(Solver):
  train_domain = ThreeFinHeatTrain
  monitor_domain = ThreeFinDesignOptMonitor

  def __init__(self, **config):
    super(ThreeFinDesignOpt, self).__init__(**config)
    heat_solver = ThreeFinHeatSolver(**config)
    self.equations = heat_solver.equations
    self.nets = heat_solver.nets

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'initialize_network_dir': './network_checkpoint_three_fin_fluid_flow_parameterized_zero_eq',
        'network_dir': './network_checkpoint_three_fin_heat_parameterized_zero_eq',
        'rec_results_cpu': True,
        'run_mode': 'eval'
        })

# read the monitor files, and perform a brute force design optimization
def DesignOpt(path, num_design, max_pressure_drop, invar_mapping, outvar_mapping):
  path += '/monitor_domain/results'
  directory = os.path.join(os.getcwd(), path)
  sys.path.append(path)
  values = []
  configs=[]
  for _, _, files in os.walk(directory):
      for file in files:
          if file.startswith("BP") & file.endswith(".csv"):
              value = []
              configs.append(file[2:-4])

              # read back pressure
              with open(os.path.join(path, file), "r") as datafile:
                  data=[]
                  reader = csv.reader(datafile, delimiter=',')
                  for row in reader:
                      columns = [row[1]]
                      data.append(columns)
                  last_row = float(data[-1][0])
                  value.append(last_row)

              # read front pressure
              with open(os.path.join(path,'FP'+file[2:]), "r") as datafile:
                  reader = csv.reader(datafile, delimiter=',')
                  data=[]
                  for row in reader:
                      columns = [row[1]]
                      data.append(columns)
                  last_row = float(data[-1][0])
                  value.append(last_row)

              # read temperature
              with open(os.path.join(path,'PT'+file[2:]), "r") as datafile:
                  data=[]
                  reader = csv.reader(datafile, delimiter=',')
                  for row in reader:
                      columns = [row[1]]
                      data.append(columns)
                  last_row = float(data[-1][0])
                  value.append(last_row)
              values.append(value)

  # perform the design optimization
  values = np.array([[values[i][1]-values[i][0], values[i][2]*273.15] for i in range(len(values))])
  indices = np.where(values[:,0]<max_pressure_drop)[0]
  values = values[indices]
  configs = [configs[i] for i in indices]
  opt_design_index = values[:,1].argsort()[0:num_design]
  opt_design_values = values[opt_design_index]
  opt_design_configs = [configs[i] for i in opt_design_index]

  # Save to a csv file
  opt_design_configs = np.array([np.array(opt_design_configs[i][1:].split('_')).astype(float) for i in range(
     num_design)])
```

```
    opt_design_configs_dict = {key: value.reshape(-1,1) for (key,value) in zip(invar_mapping, opt_design_configs.
        T)}
    opt_design_values_dict = {key: value.reshape(-1,1) for (key,value) in zip(outvar_mapping, opt_design_values.T)
        }
    opt_design = {**opt_design_configs_dict, **opt_design_values_dict}
    dict_to_csv(opt_design,'optimal_design')
    print('Finished design optimization!')

if __name__ == '__main__':
    ctr = SimNetController(ThreeFinDesignOpt)
    ctr.run()
    DesignOpt(path, num_design, max_pressure_drop, invar_mapping, outvar_mapping)
```

Listing 132: The Optimizer Script

The above script can then be run using by executing the following command.

```
python three_fin_optimizer_zero_eq_parameterized.py
```

16.6 Results and Post-processing

The output of the optimization script is stored in a .csv file which ranks 20 designs from the sample with the most optimal desing ranked first. The design parameters for the optimal heat sink for this problem are: $h_{centralfin} = 0.4$, $h_{sidefins} = 0.4$, $l_{centralfin} = 0.83$, $l_{sidefins} = 1.0$, $t_{centralfin} = 0.15$, $t_{sidefins} = 0.15$. The above design has a pressure drop of 2.46 and a peak temperature of 76.23 $°C$ (Figure 74)

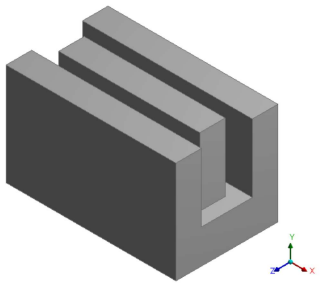


Figure 74: Three Fin geometry after optimization

Table 7 represents the computed pressure drop and peak temperature for the OpenFOAM single geometry and SimNet single and parameterized geometry runs. It is evident that the results for the parameterized model are close to those of a single geometry model, showing its good accuracy.

Table 7: A comparison for the OpenFOAM and SimNet results

Property	OpenFOAM Single Run	Single Run	Parameterized Run
Pressure Drop (Pa)	2.195	2.063	2.016
Peak Temperature $(°C)$	72.68	76.10	77.41

By parameterizing the geometry, SimNet significantly accelerates design optimization when compared to traditional solvers, which are limited to single geometry simulations. For instance, 3 values (two end values of the range and a middle value) per design variable would result in 3^6 = 729 single geometry runs. The total compute time required by OpenFOAM and SimNet for this design optimization task is reported in Table 8. It can be seen that for 729 OpenFOAM runs, it would result in 4099 wall-hrs of compute time while the neural network takes only 120 wall-hrs of compute time. Large number of design variables or their values would only magnify the difference in the time taken for two approaches.

Table 8: Total compute time needed for OpenFOAM and SimNet for the three fin heat sink design optimization

Solver	OpenFOAM	SimNet
Compute Time (hrs)	4099	120

Note: The SimNet calculations were done using 4 Nvidia V100 GPUs. The OpenFOAM calculations were done using 20 processors.

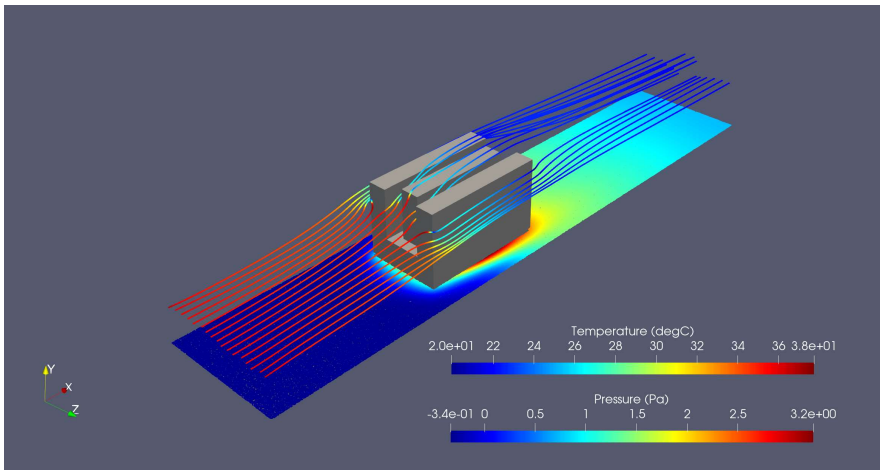


Figure 75: Streamlines colored with pressure and temperature profile in the fluid for optimal three fin geometry

Here, the 3-Fin heatsink was solved for arbitrary heat properties chosen such that the coupled conjugate heat transfer solution was possible. However, such approach causes issues when the conductivities are orders of magnitude different at the interface. We will revisit the conjugate heat transfer problem in tutorial 18 where we will solve the NVSwitch heatsink with real heat properties and also show how to use gPC for constructing surrogates for heatsink design optimization.

17 Case Study: FPGA Heat Sink with Laminar Flow for Single Geometry: Comparisons of Network Architectures, Optimizers and other schemes in SimNet

17.1 Introduction

In this example we show how some of the features in SimNet apply for a complicated FPGA heat sink design and solve the conjugate heat transfer. In this tutorial you would learn the following:

1. How to use Fourier Networks for complicated geometries with sharp gradients
2. How to solve problem with symmetry using symmetry boundary conditions
3. How to formulate velocity field as a vector potential (Exact continuity feature)
4. How different features and architectures in SimNet perform on a problem with complicated geometry

Prerequisites

This tutorial is very similar to the conjugate heat transfer problem that we have seen in the tutorials 13 and 16. We strongly recommend you to visit these tutorials (especially tutorial 13) for the details on the geometry generation, boundary conditions and monitor and validation domains. This tutorial would skip the description for the above stated processes and would instead focus more on the implementation of different features and the case study.

17.2 Problem Description

The geometry of the FPGA heatsink can be seen in figure 76. This particular geometry is challenging to simulate due to the thin closely spaced fins that causes sharp gradients which are particularly difficult to learn using regular fully connected neural network (slow convergence).

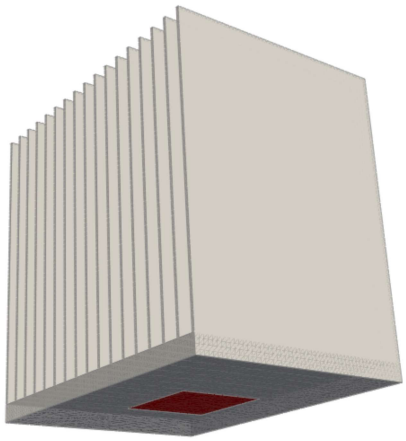


Figure 76: The FPGA heat sink geometry

We will be solving the conjugate heat transfer problem for the above geometry at Re=50. The dimensions of the geometry, as modeled in SimNet, are summarized in the Table 9:

Table 9: FPGA Dimensions

Dimension	Value
Heat Sink Base (l x b x h)	0.65 x 0.875 x 0.05
Fin dimension (l x b x h)	0.65 x 0.0075 x 0.8625
Heat Source (l x b)	0.25 x 0.25
Channel (l x b x h)	5.0 x 1.125 x 1.0

All the dimensions are scaled such that the channel height is 1 m. The temperature is scaled according to $\theta = T/273.15 - 1.0$. The channel walls are treated as adiabatic and the interface boundary conditions are applied at the fluid-solid interface. Other flow and thermal parameters are described in the Table 10

Table 10: Fluid and Solid Properties

Property	Fluid	Solid
Inlet Velocity (m/s)	1.0	NA
Density (kg/m^3)	1.0	1.0
Kinematic Viscosity (m^2/s)	0.02	NA
Thermal Diffusivity (m^2/s)	0.02	0.1
Thermal Conductivity $(W/m.K)$	1.0	1.0
Inlet Temperature (K)	273.15	NA
Heat Source Temperature Gradient (K/m)	409.725	NA

Case Setup

As suggested earlier, the case setup for this problem is very similar to the problem described in tutorial 13. Like tutorial 13 we will have 3 separate scripts for this problem for the domain definition, flow solver and heat solver. We will skip over the definitions of these files and would discuss only the major features introduced in this chapter.

Note: All the relevant domain, flow and heat solver files for this problem using various versions of features can be found at `examples/fpga/`.

17.3 Solver using Fourier Network Architecture

As described in the Theory section in tutorial 1, in SimNet, we can overcome the spectral bias of the neural networks by using the Fourier Networks. These networks have shown a significant improvement in results over the regular fully connected neural networks due to their ability to capture sharp gradients.

We do not need to make any special changes to the way the geometry and train domain are created while making changes to the neural network architectures. This also means the architecture is independent of the physics or parameterization being solved and can be applied to any other class of problems covered in the User Guide.

A note on frequencies: One main parameters of this networks are how we choose the frequencies. In SimNet we can choose from the spectrum we want to sample (full/partial/axis) and the number of frequencies in the spectrum. The optimal number of frequencies is an active field of research as one often needs to balance the accuracy benefits and the computational expense added due to use of extra Fourier features. For FPGA problem, we find that choosing 25 frequencies for laminar flow and 35 for turbulent flow gives a good balance between the two.

Moreover, for large domains where the length scale of the object of interest is significantly smaller than the max length in the domain (eg. bluff body simulations), we have found that normalizing the frequencies by that max length initializes the network better and speeds up convergence. Such modification can be achieved by setting the frequencies as `i/max_length` instead of just `i`. For the FPGA problem here, the `max_length` in the domain is the channel length in x-direction i.e. 5.0.

Below we show the solver file for the parameterized FPGA flow field simulation (`max_length = 5.0`), using the Fourier network architecture used with partial spectrum (axis and diagonal), 25 frequencies and with frequency normalization.

```
# import domain
from fpga_domain_parameterized import FPGAFlowTrain, FPGAFlowMonitor, FPGAFlowVal, FPGAFlowInference, nu, rho

# import SimNet library
from simnet.solver import Solver
from simnet.PDES.navier_stokes import IntegralContinuity, NavierStokes
from simnet.controller import SimNetController
from simnet.architecture.fourier_net import FourierNetArch

class FPGAFlowSolver(Solver):
  train_domain = FPGAFlowTrain
  monitor_domain = FPGAFlowMonitor
  val_domain = FPGAFlowVal
  inference_domain = FPGAFlowInference
  arch = FourierNetArch
```

```
  def __init__(self, **config):
    super(FPGAFlowSolver, self).__init__(**config)
    self.frequencies = ('axis,diagonal', [i/5. for i in range(25)])
    self.frequencies_params = ('axis,diagonal', [i/5. for i in range(20)])

    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z', 'HS_height', 'HS_length'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_fpga_fluid_flow_parameterized_fourier_net',
        'rec_results_cpu': True,
        'max_steps': 750000,
        'decay_steps': 7500
        })

if __name__ == '__main__':
  ctr = SimNetController(FPGAFlowSolver)
  ctr.run()
```

Listing 133: Using Fourier Networks for the FPGA problem

17.4 Leveraging Symmetry of the Problem

Whenever we have a symmetric geometry and we expect the variable fields to be symmetric, we can use symmetry boundary conditions about the plane or axis symmetry to minimize the computational expense of modeling the entire geometry. For the FPGA heat sink, we have such a plane of symmetry in the z-plane (Figure 77). The symmetry boundary conditions can be referred in Section 1.5.8 of the tutorial 1. Simulating the FPGA problem using symmetry, we achieve about 33% reduction in training time, compared to a training on the full domain.

For the FPGA problem where the plane of symmetry is z-plane, the boundary conditions stated in Section 1.5.8 can be translated to the following:

1. Variables which are odd functions w.r.t. `z` coordinate axis: `'w'`. Hence on symmetry plane `'w'=0`.
2. Variables which are even functions w.r.t. `z` coordinate axis: `'u'`, `'v'` components of velocity vector and scalar quantities like `'p'`, `'theta_s'` , `'theta_f'`. Hence on symmetry plane we set their normal derivative to 0. Eg. `'u__z'=0`.

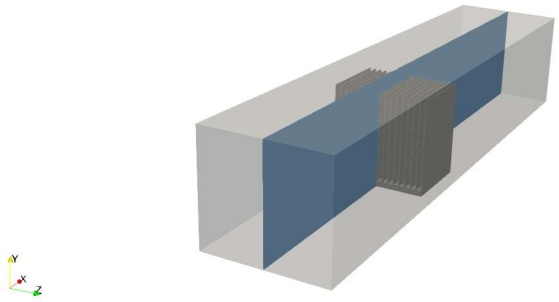


Figure 77: The FPGA heat sink with plane of symmetry

Below we show only the symmetry boundary conditions in the flow and heat training domains as rest other portion of train domain remain the same. (Full domain file can be accessed at `examples/fpga/`)

```
class FPGAFlowTrain(TrainDomain):
  def __init__(self, **config):
    super(FPGAFlowTrain, self).__init__()
    ...
    # symmetry BC
```

```
    symmetry = geo.boundary_bc(outvar_sympy={'w': 0, 'u__z': 0, 'v__z': 0, 'p__z': 0},
                               batch_size_per_area=500,
                               batch_per_epoch=5000,
                               criteria = Eq(z, channel_origin[2]+channel_dim[2]/2.))
    self.add(symmetry, name="SymmetricChannel")
    ...

class FPGAHeatTrain(TrainDomain):
  def __init__(self, **config):
    super(FPGAHeatTrain, self).__init__()
    ...
    # symmetry BC
    symmetrySolid = fpga.boundary_bc(outvar_sympy={'theta_s__z': 0},
             batch_size_per_area=500,
             criteria = Eq(z, channel_origin[2]+channel_dim[2]/2.),
             batch_per_epoch=5000)
    self.add(symmetrySolid, name="SymmetricChannelSolid")

    symmetryFluid = geo.boundary_bc(outvar_sympy={'theta_f__z': 0},
                                    batch_size_per_area=500,
                                    criteria = Eq(z, channel_origin[2]+channel_dim[2]/2.),
           batch_per_epoch=5000)
    self.add(symmetryFluid, name="SymmetricChannelFluid")
    ...
```

Listing 134: Using Symmetry boundary conditions for the FPGA problem

17.5 Imposing Exact Continuity

As described in the Theory section in tutorial 1, in SimNet, we can define the velocity field as a vector potential such that it is divergence free and satisfies continuity automatically. We can use this formulation for any class of flow problems covered in this User Guide regardless of the network architecture. However, we have found it to be most effective when using fully connected networks.

Below code shows the flow solver file for using exact continuity feature. No specific changes need to made to the train domain definition to use this feature.

```
# import domain
from fpga_domain import FPGAFlowTrain, FPGAFlowMonitor, FPGAFlowVal, nu, rho

from simnet.solver import Solver
from simnet.PDES.navier_stokes import NavierStokes, IntegralContinuity, Curl
from simnet.controller import SimNetController

# Define neural network
class FPGAFlowSolver(Solver):
  train_domain     = FPGAFlowTrain
  monitor_domain   = FPGAFlowMonitor
  val_domain       = FPGAFlowVal

  def __init__(self, **config):
    super(FPGAFlowSolver, self).__init__(**config)
    c = Curl(('a','b','c'), ('u','v','w'))
    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node()
                      + IntegralContinuity(dim=3).make_node()
                      + c.make_node())
    flow_net  = self.arch.make_node(name='flow_net',
                                    inputs=['x','y','z'],
                                    outputs=['a','b','c','p'])
    self.nets = [flow_net]

  @classmethod # Explain This
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_fpga_fluid_flow_exact_continuity',
        'rec_results': True,
        'rec_results_freq': 1000,
        })

if __name__ == '__main__':
  ctr = SimNetController(FPGAFlowSolver)
  ctr.run()
```

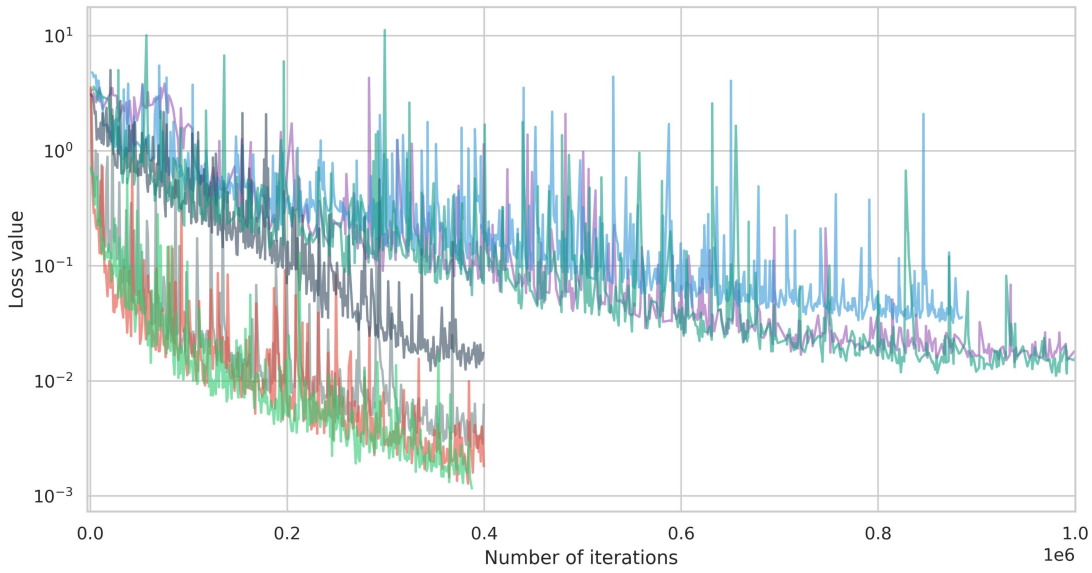
Listing 135: Defining velocity field as a vector potential

17.6 Results, Comparisons, and Summary

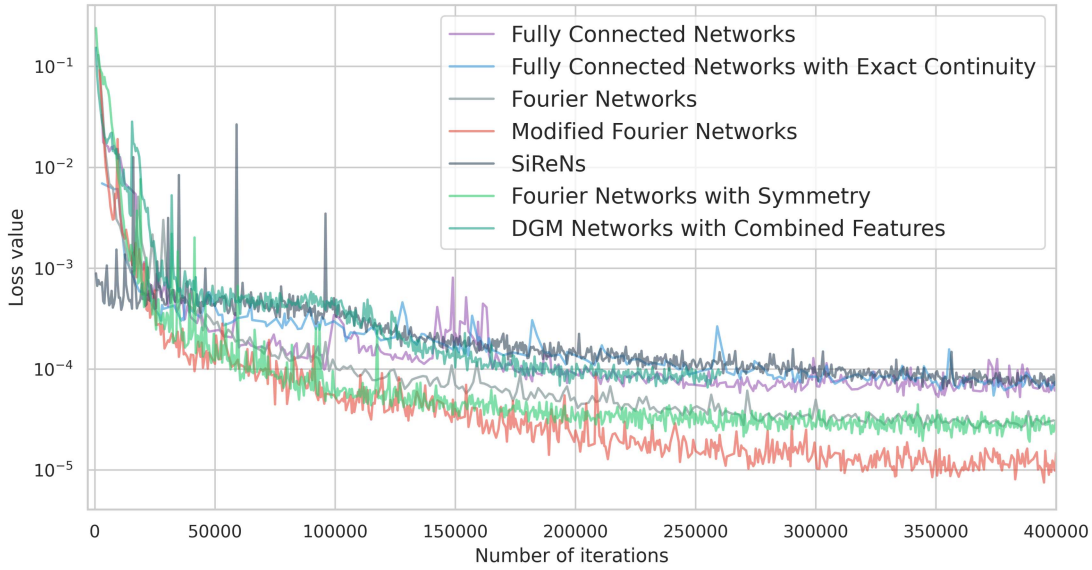
We have discussed several features in this chapter which might be a bit overwhelming. In the Table 11, we summarize these features and their applications and in Table 12, we summarize the important results of these features on this FPGA problem. Also, in Figure 78, we provide a comparison for the loss values from different runs.

Table 11: Summary of features introduced in this tutorial

Feature	Applicability to other problems	Comments
Fourier Networks	Applicable to all class of problems	Shown to be highly effective for problems involving sharp gradients. Modified Fourier network found to improve the performance one step further.
Symmetry	Applicable to all problems with a plane/axis of symmetry	Reduces the computational domain to half leading to significant speedup (33% reduction in training time compared to full domain)
Exact Continuity	Applicable to incompressible flow problems requiring solution to Navier Stokes equation	Gives better satisfaction of the continuity equation than Velocity-pressure formulation. Found to work best with standard fully connected networks. Also improves the accuracy of results in Fourier networks.
SiReNs	Applicable to all class of problems	Shown to be effective for problems with sharp gradients. However, we find that it does not outperform the Fourier Networks in terms of accuracy.
DGM Networks with Global LR annealing, Global Adaptive Activations and Halton Sequences	Applicable to all class of problems	Improves accuracy compared to regular fully connected networks.



(a) flow



(b) heat

Figure 78: A comparison for the loss values from different runs as listed in Table 12

Table 12: Comparison of pressure drop and peak temperatures from various runs

Case Description	$P_{drop}\ (Pa)$	$T_{peak}\ (^{\circ}C)$
SimNet: Fully Connected Networks	29.24	77.90
SimNet: Fully Connected Networks with Exact Continuity	28.92	90.63
SimNet: Fourier Networks	29.19	77.08
SimNet: Modified Fourier Networks	29.23	80.96
SimNet: SiReNs	29.21	76.54
SimNet: Fourier Networks with Symmetry	29.14	78.56
SimNet: DGM Networks with Global LR annealing, Global Adaptive Activations, and Halton Sequences	29.10	76.86
OpenFOAM Solver	28.03	76.67
Commercial Solver	28.38	84.93

18 Industrial Heat Sink simulations

18.1 Introduction

In this tutorial, we will use SimNet to conduct a thermal simulation of NVIDIA's NVSwitch heatsink. You will learn the following:

1. How to use hFTB algorithm to solve conjugate heat transfer problems
2. How to make a script to evaluate results on mesh grids
3. How to build a gPC-Based Surrogate via Transfer Learning

Prerequisites

This tutorial assumes you have completed tutorial 6 on transient Navier-Stokes via moving time window as well as the tutorial 13 on conjugate heat transfer.

18.2 Problem Description

Here, we solve the conjugate heat transfer problem of NVIDIA's NVSwitch heat sink as shown in 79. Similar to the previous FPGA problem, the heat sink is placed in a channel with inlet velocity similar to its operating conditions. This case differs from our previous conjugate heat transfer problems because we will be using the real heat properties for atmospheric air and copper as the heat sink material.

Using real heat properties causes an issue on the interface between the solid and fluid because the conductivity is around 4 orders of magnitude different (Air: 0.0261 $W/m.K$ and Copper: 385 $W/m.K$). If attempting to solve in a fully coupled manner as before this will cause issues with convergence. To remedy this, we have implemented a static conjugate heat transfer approached referred to as heat transfer coefficient forward temperature backward or hFTB [34]. This method works by iteratively solving for the heat transfer in the fluid and solid where they are one-way coupled. Using the hFTB method, we assign Robin boundary conditions on the solid interface and Dirichlet boundaries for the fluid. The simulation starts by giving an initial guess for the solid temperature and uses a hyper parameter h for the Robin boundary conditions. We give a description of the algorithm here 1 and refer the readers to more complete description here [34].

Algorithm 1: hFTB algorithm

Solve fluid with interface Dirichlet=INITIAL_TEMP;
for *i :=1 to NUMBER_CYCLES* **do**
    AMBIENT_TEMP = FLUID_TEMP - FLUID_FLUX / h;
    Solve Solid with interface Robin=SOLID_FLUX - h(SOLID_TEMP - AMBIENT_TEMP);
    Solve Fluid with Dirichlet=SOLID_TEMP;
end

18.3 Case Setup

The case setup for this problem is similar to the FPGA and three fin examples (covered in tutorials 13 and 17) however we will construct multiple train domains to implement the hFTB method. We will also show a script that can be used to evaluate the results on a mesh grid.

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

18.3.1 Defining Domain

In this case setup we will skip over several sections of the code and only focus on the portions related to the hFTB algorithm. We will assume that the reader is familiar with how to set up the flow simulation from previous tutorials. We also will not go into the details about the geometry construction and assume we can get all relevant information from the module `examples/limerock/limerock_hFTB/stl_to_geo.py`. We will begin the code description by defining the parameters of the simulation and importing all needed modules.

```
# import geometry
from stl_to_geo import LimeRock
```

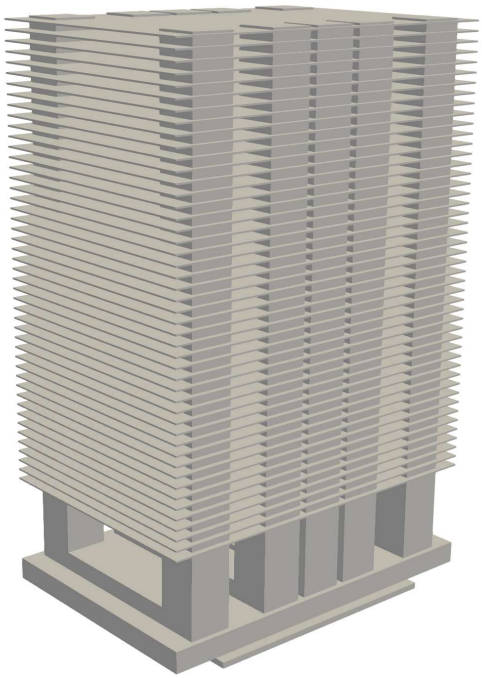


Figure 79: NVSwitch heat sink geometry

```
# import SimNet library
from sympy import Symbol, Eq, tanh, Abs, Or
import numpy as np
import tensorflow as tf
from simnet.dataset import TrainDomain, ValidationDomain, MonitorDomain, InferenceDomain
from simnet.data import Validation, Monitor, Inference
from simnet.sympy_utils.geometry_3d import Box, Channel, Plane
from simnet.csv_utils.csv_rw import csv_to_dict

# make limerock
limerock = LimeRock()

##############
# Real Params
##############
# fluid params
fluid_viscosity = 1.84e-05 # kg/m-s
fluid_density = 1.1614 # kg/m3
fluid_specific_heat = 1005 # J/(kg K)
fluid_conductivity = 0.0261 # W/(m K)

# copper params
copper_density = 8930 # kg/m3
copper_specific_heat = 385 # J/(kg K)
copper_conductivity = 385 # W/(m K)

# boundary params
inlet_velocity = 5.7 # m/s
inlet_temp = 0 # K

# source
source_term = 2127.71 # K/m
source_origin = (-0.061667, -0.15833, limerock.geo_bounds_lower[2])
source_dim = (0.1285, 0.31667, 0)

#################
# Non dim params
#################
length_scale = 0.0575 # m
velocity_scale = 5.7 # m/s
time_scale = length_scale/velocity_scale # s
density_scale = 1.1614 # kg/m3
mass_scale = density_scale*length_scale**3 # kg
pressure_scale = mass_scale / (length_scale * time_scale**2) # kg / (m s**2)
```

```
temp_scale = 273.15 # K
watt_scale = (mass_scale * length_scale**2) / (time_scale**3) # kg m**2 / s**3
joule_scale = (mass_scale * length_scale**2) / (time_scale**2) # kg * m**2 / s**2

##############################
# Nondimensionalization Params
##############################
# fluid params
nd_fluid_viscosity      = fluid_viscosity / (length_scale**2 / time_scale) # need to divide by density to get
    previous viscosity
nd_fluid_density        = fluid_density / density_scale
nd_fluid_specific_heat  = fluid_specific_heat / (joule_scale / (mass_scale * temp_scale))
nd_fluid_conductivity   = fluid_conductivity / (watt_scale / (length_scale * temp_scale))
nd_fluid_diffusivity    = nd_fluid_conductivity / (nd_fluid_specific_heat * nd_fluid_density)

# copper params
nd_copper_density       = copper_density / (mass_scale / length_scale**3)
nd_copper_specific_heat = copper_specific_heat / (joule_scale / (mass_scale * temp_scale))
nd_copper_conductivity  = copper_conductivity / (watt_scale / (length_scale * temp_scale))
nd_copper_diffusivity   = nd_copper_conductivity / (nd_copper_specific_heat * nd_copper_density)

# boundary params
nd_inlet_velocity  = inlet_velocity/velocity_scale
nd_volumetric_flow = limerock.inlet_area * nd_inlet_velocity
nd_inlet_temp      = inlet_temp / temp_scale
nd_source_term     = source_term / (temp_scale / length_scale)
```

Listing 136: Defining simulation parameters

Note: We non-dimensionalize all parameters so that the scales for velocity, temperature, and pressure are roughly in the range 0-1. Such non dimensionalization trains the Neural network more efficiently.

Now we will set up a train domain to only solve for the temperature in the fluid given a Dirichlet boundary condition on the solid. This will be the first stage of the hFTB method.

```
class Cycle1hFTBTrain(TrainDomain):
  name = 'cycle_1'
  nr_iterations = 1

  def __init__(self, **config):
    super(Cycle1hFTBTrain, self).__init__()
    scale_batch = 4

    # get initial temp for hFTB algorithm
    initial_theta_f = config['config'].initial_t # This can be any temp but 1.0 is a good guess for this
     problem

    # inlet
    inletBC = limerock.inlet.boundary_bc(outvar_sympy={'theta_f': nd_inlet_temp},
                                         batch_size_per_area=50//scale_batch,
                                         lambda_sympy={'lambda_theta_f': 1000.0},
                                         batch_per_epoch=2000*scale_batch,
                                         criteria=Eq(x, limerock.geo_bounds_lower[0]))
    self.add(inletBC, name="Inlet_Cycle1")
    print("made inlet")

    # outlet
    outletBC = limerock.outlet.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                           batch_size_per_area=50//scale_batch,
                                           lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                           batch_per_epoch=2000*scale_batch,
                                           criteria=Eq(x, limerock.geo_bounds_upper[0]))
    self.add(outletBC, name="Outlet_Cycle1")
    print("made outlet")

    # channel walls insulating
    walls = limerock.geo.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                     batch_size_per_area=50//scale_batch,
                                     lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                     criteria=Eq(y, limerock.geo_bounds_lower[1])|Eq(z, limerock.
     geo_bounds_lower[2])|Eq(y, limerock.geo_bounds_upper[1])|Eq(z, limerock.geo_bounds_upper[2]),
                                     batch_per_epoch=2000*scale_batch)
    self.add(walls, name="ChannelWalls_Cycle1")
    print("made channel walls")

    # flow interior low res away from heat sink
    interiorF_lr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                            bounds=limerock.geo_bounds,
                                            batch_size_per_area=120//scale_batch,
```

```
                                        lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                        criteria=(x < limerock.heat_sink_bounds[0]) | (x > limerock.
    heat_sink_bounds[1]),
                                        batch_per_epoch=2000*scale_batch)
    self.add(interiorF_lr, name="FlowInterior_LR_Cycle1")
    print("made lr flow interior")

    # flow interiror high res near heat sink
    interiorF_hr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                        bounds=limerock.geo_hr_bounds,
                                        batch_size_per_area=750//scale_batch,
                                        lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                        batch_per_epoch=2000*scale_batch)
    self.add(interiorF_hr, name="FlowInterior_HR_Cycle1")
    print("made hr flow interior")

    # fluid solid interface
    interface = limerock.geo_solid.boundary_bc(outvar_sympy={'theta_f': initial_theta_f},
                                            batch_size_per_area=50//scale_batch,
                                            lambda_sympy={'lambda_theta_f': 100.0},
                                            criteria=z>limerock.geo_bounds_lower[2],
                                            batch_per_epoch=2000*scale_batch)
    self.add(interface, name="Interface_Cycle1")
    print("made solid flow interface")
```

Listing 137: Defining Train Domain

After getting this initial solution for the temperature in the fluid we solve for the main loop of the hFTB algorithm. Now we will solve for both the fluid and solid in a one way coupled manner. The Robin boundary conditions for the solid are coming from the previous iteration of the fluid solution.

```
class CycleNhFTBTrain(TrainDomain):
  name = 'cycle_n'
  nr_iterations = 10

  def __init__(self, **config):
    super(CycleNhFTBTrain, self).__init__()
    scale_batch = 4

    # inlet
    inletBC = limerock.inlet.boundary_bc(outvar_sympy={'theta_f': nd_inlet_temp},
                                        batch_size_per_area=50//scale_batch,
                                        lambda_sympy={'lambda_theta_f': 1000.0},
                                        batch_per_epoch=2000*scale_batch,
                                        criteria=Eq(x, limerock.geo_bounds_lower[0]))
    self.add(inletBC, name="Inlet")
    print("made inlet")

    # outlet
    outletBC = limerock.outlet.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                         batch_size_per_area=50//scale_batch,
                                         lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                         batch_per_epoch=2000*scale_batch,
                                         criteria=Eq(x, limerock.geo_bounds_upper[0]))
    self.add(outletBC, name="Outlet")
    print("made outlet")

    # channel walls insulating
    walls = limerock.geo.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                    batch_size_per_area=50//scale_batch,
                                    lambda_sympy={'lambda_normal_gradient_theta_f': 1.0},
                                    criteria=Eq(y, limerock.geo_bounds_lower[1])|Eq(z, limerock.
    geo_bounds_lower[2])|Eq(y, limerock.geo_bounds_upper[1])|Eq(z, limerock.geo_bounds_upper[2]),
                                    batch_per_epoch=2000*scale_batch)
    self.add(walls, name="ChannelWalls")
    print("made channel walls")

    # flow interior low res away from heat sink
    interiorF_lr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                        bounds=limerock.geo_bounds,
                                        batch_size_per_area=120//scale_batch,
                                        lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                        criteria=(x < limerock.heat_sink_bounds[0]) | (x > limerock.
    heat_sink_bounds[1]),
                                        batch_per_epoch=2000*scale_batch)
    self.add(interiorF_lr, name="FlowInterior_LR")
    print("made lr flow interior")

    # flow interiror high res near heat sink
```

```
    interiorF_hr = limerock.geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                            bounds=limerock.geo_hr_bounds,
                                            batch_size_per_area=750//scale_batch,
                                            lambda_sympy={'lambda_advection_diffusion': 1000.0},
                                            batch_per_epoch=2000*scale_batch)
    self.add(interiorF_hr, name="FlowInterior_HR")
    print("made hr flow interior")

    # diffusion dictionaries
    diffusion_outvar_sympy = {'diffusion_theta_s': 0,
                              'compatibility_theta_s_x_y': 0,
                              'compatibility_theta_s_x_z': 0,
                              'compatibility_theta_s_y_z': 0,
                              'integrate_diffusion_theta_s_x': 0,
                              'integrate_diffusion_theta_s_y': 0,
                              'integrate_diffusion_theta_s_z': 0}
    diffusion_lambda_sympy = {'lambda_diffusion_theta_s': 1000000.0,
                              'lambda_compatibility_theta_s_x_y': 1.0,
                              'lambda_compatibility_theta_s_x_z': 1.0,
                              'lambda_compatibility_theta_s_y_z': 1.0,
                              'lambda_integrate_diffusion_theta_s_x': 1.0,
                              'lambda_integrate_diffusion_theta_s_y': 1.0,
                              'lambda_integrate_diffusion_theta_s_z': 1.0}

    # solid interior
    interiorS = limerock.geo_solid.interior_bc(outvar_sympy=diffusion_outvar_sympy,
                                               bounds=limerock.geo_hr_bounds,
                                               batch_size_per_area=5000//scale_batch,
                                               lambda_sympy=diffusion_lambda_sympy,
                                               batch_per_epoch=2000*scale_batch)
    self.add(interiorS, name="SolidInterior")
    print("made solid interior")

    # limerock base
    sharpen_tanh = 60.0
    source_func_xl = (tanh(sharpen_tanh*(x - source_origin[0])) + 1.0)/2.0
    source_func_xh = (tanh(sharpen_tanh*((source_origin[0]+source_dim[0]) - x)) + 1.0)/2.0
    source_func_yl = (tanh(sharpen_tanh*(y - source_origin[1])) + 1.0)/2.0
    source_func_yh = (tanh(sharpen_tanh*((source_origin[1]+source_dim[1]) - y)) + 1.0)/2.0
    gradient_normal = nd_source_term * source_func_xl * source_func_xh * source_func_yl * source_func_yh
    limerock_base = limerock.geo_solid.boundary_bc(outvar_sympy={'normal_gradient_flux_theta_s':
     gradient_normal},
                                                   lambda_sympy={'lambda_normal_gradient_flux_theta_s': 10.0},
                                                   batch_size_per_area=500//scale_batch, # NOTE I put this a
     guess for the batch size and am still looking at it
                                                   criteria=Eq(z,limerock.geo_bounds_lower[2]),
                                                   batch_per_epoch=2000*scale_batch)
    self.add(limerock_base, name="HeatSinkBase")
    print("made base source")

    # fluid solid interface
    interface = limerock.geo_solid.boundary_bc(outvar_sympy={'dirichlet_theta_s_theta_f': 0,
                                                             'robin_theta_s': 0},
                                               batch_size_per_area=50//scale_batch,
                                               lambda_sympy={'lambda_dirichlet_theta_s_theta_f': 100.0,
                                                             'lambda_robin_theta_s': 1.0},
                                               criteria=z>limerock.geo_bounds_lower[2],
                                               batch_per_epoch=2000*scale_batch)
    self.add(interface, name="Interface")
    print("made solid flow interface")
```

Listing 138: Defining Train Domain

18.3.2 Sequence Solver

Now we setup the solver. Similar to the moving time window implementation (tutorial 6), we construct a separate neural network that stores the thermal solution from the previous cycles fluid solution. We also define a `custom_update_op` that will restart the learning rate decay and update networks weights after each cycle. We also add an option to allow for gradient aggregation with the default set to 2. This can be used to increase the batch size. We suggest that this problem is either run on 4 GPUs or this gradient aggregation is set to 8. Details on running with multi-GPUs and multi-nodes can be found in tutorial 19 and the details on using gradient aggregation can be found in tutorial 12.

```
# import domain
from limerock_domain import CyclelhFTBTrain, CycleNhFTBTrain, LRHeatMonitor, LRHeatInference, limerock,
    nd_fluid_viscosity, nd_fluid_density, nd_fluid_diffusivity, nd_copper_conductivity, nd_fluid_conductivity,
    nd_copper_diffusivity
```

```
# import vector flux diffusion equations
from flux_diffusion import FluxDiffusion, FluxGradNormal, FluxIntegrateDiffusion, FluxRobin, Dirichlet

# import SimNet library
from simnet.solver import Solver
from simnet.PDES.navier_stokes import GradNormal
from simnet.PDES.advection_diffusion import AdvectionDiffusion
from simnet.controller import SimNetController
from simnet.architecture.fourier_net import FourierNetArch
from simnet.node import Node
from simnet.variables import Key, Variables
from simnet.config import str2bool
from simnet.optimizer import AdamOptimizerGradAgg
grad_agg_freq = 2  # number of gradient aggregations

class LRHeatSolver(Solver):
  seq_train_domain = [Cycle1hFTBTrain, CycleNhFTBTrain]
  monitor_domain = LRHeatMonitor
  arch = FourierNetArch
  optimizer = AdamOptimizerGradAgg

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

    self.arch.frequencies = ('axis', [i/5. for i in range(50)])

    # configs for the hFTB algorithm
    h = config['config'].h # This can be arbitrarily positive number
    initial_theta_f = config['config'].initial_t # This can be any temp that is in the range possible solutions,
      say between 0-1.

    # make list of equations
    self.equations = (AdvectionDiffusion(T='theta_f', rho=nd_fluid_density, D=nd_fluid_diffusivity, dim=3, time
     =False).make_node(stop_gradients=['u', 'v', 'w'])
                    + GradNormal('theta_f', dim=3, time=False).make_node()
                    + FluxDiffusion(D=nd_copper_diffusivity).make_node()
                    + FluxIntegrateDiffusion().make_node(stop_gradients=['flux_theta_s_x', 'flux_theta_s_y', '
     flux_theta_s_z'])
                    + FluxGradNormal().make_node()
                    + FluxRobin(theta_f_conductivity=nd_fluid_conductivity, theta_s_conductivity=
     nd_copper_conductivity, h=h).make_node(stop_gradients=['theta_f_prev_step','theta_f_prev_step__x','
     theta_f_prev_step__y','theta_f_prev_step__z'])
                    + Dirichlet(lhs='theta_f', rhs='theta_s').make_node(stop_gradients=['theta_s']))

    # make networks
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    solid_heat_net = self.arch.make_node(name='solid_heat_net',
                                         inputs=['x', 'y', 'z'],
                                         outputs=['theta_s'])
    flux_heat_net = self.arch.make_node(name='flux_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['flux_theta_s_x',
                                                 'flux_theta_s_y',
                                                 'flux_theta_s_z'])
    flow_heat_net = self.arch.make_node(name='flow_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['theta_f'])
    flow_heat_net_prev_step = self.arch.make_node(name='flow_heat_net_prev_step',
                                                  inputs=['x', 'y', 'z'],
                                                  outputs=['theta_f_prev_step'])
    self.nets = [flow_net, solid_heat_net, flux_heat_net, flow_heat_net, flow_heat_net_prev_step]

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

    # make update op that sets weights from_flow_net to flow_net_prev_step
    prev_assign_step = [zero_step_op]
    flow_heat_net_variables = [v for v in tf.trainable_variables() if 'flow_heat_net/' in v.name]
    flow_heat_net_prev_step_variables = [v for v in tf.trainable_variables() if 'flow_heat_net_prev_step' in v.
     name]
    for v, v_prev_step in zip(flow_heat_net_variables, flow_heat_net_prev_step_variables):
      prev_assign_step.append(tf.assign(v_prev_step, v))
    prev_assign_step = tf.group(*prev_assign_step)
    return prev_assign_step

  @classmethod
```

```
  def add_options(cls, group):
    group.add_argument('--initial_t',
                       help='initial tempurature',
                       type=float,
                       default=0.05)
    group.add_argument('--h',
                       help='H values',
                       type=float,
                       default=500.0)

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'initialize_network_dir': './network_checkpoint_limerock_fluid_flow_real',
        'network_dir': './network_checkpoint_limerock_heat_hFTB',
        'added_config_dir': 'initial_t,h',
        'rec_results_cpu': True,
        'max_steps': 150000*grad_agg_freq,
        'decay_steps': 1500*grad_agg_freq,
        'rec_results_freq': 1000,
        'start_lr': 1e-3,
        'grad_agg_freq': grad_agg_freq
        })

if __name__ == '__main__':
  ctr = SimNetController(LRHeatSolver)
  ctr.run()
```

Listing 139: Defining the Solver

18.3.3 Mesh Grid Evaluation Script

Evaluating with the Inference Domain only allows the results to be saved as a .vti point cloud. Sometimes it is beneficial to visualize the results on a mesh. The following scripts shows how this is possible using the `stream` method. The script starts by defining a solver class like before however we are now giving a specific network checkpoint to restore from (`./network_checkpoint_limerock_heat_hFTB_2/initial_t.0.05/h.500.0/cycle_n_0009`). Next we construct an function `thermal_stream` that will allow us to evaluate results on arbitrary points. The remainder of the script is devoted to making a mesh grid to evaluate on and saving the results. Note that we use the SDF to zero the temperature evaluations outside the solid.

```
# import domain
from stl_to_geo import LimeRock

# import needed packages
import sys
from sympy import Symbol
import numpy as np
from evtk.hl import imageToVTK

# import SimNet
from simnet.solver import Solver
from simnet.sympy_utils.geometry_3d import Box, Sphere
from simnet.dataset import TrainDomain
from simnet.controller import SimNetController
from simnet.node import Node
from simnet.variables import Variables, Key
from simnet.config import str2bool
from simnet.sympy_utils.numpy_printer import np_lambdify
from simnet.architecture.fourier_net import FourierNetArch

# make limerock
limerock = LimeRock()

# construct solver class for loading neural network simulation
class LRSolver(Solver):
  train_domain = TrainDomain # blank domain
  arch = FourierNetArch

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

    self.arch.frequencies = ('axis', [i/5. for i in range(50)])

    # make networks
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
```

```
                                    outputs=['u', 'v', 'w', 'p'])
    solid_heat_net = self.arch.make_node(name='solid_heat_net',
                                         inputs=['x', 'y', 'z'],
                                         outputs=['theta_s'])
    flux_heat_net = self.arch.make_node(name='flux_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['flux_theta_s_x',
                                                 'flux_theta_s_y',
                                                 'flux_theta_s_z'])
    flow_heat_net = self.arch.make_node(name='flow_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['theta_f'])
    flow_heat_net_prev_step = self.arch.make_node(name='flow_heat_net_prev_step',
                                                  inputs=['x', 'y', 'z'],
                                                  outputs=['theta_f_prev_step'])
    self.nets = [flow_net, solid_heat_net, flux_heat_net, flow_heat_net, flow_heat_net_prev_step]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_limerock_heat_hFTB_2/initial_t.0.05/h.500.0/cycle_n_0009',
        'rec_results_cpu': True,
        })

# make controler to start streaming results
ctr = SimNetController(LRSolver)

# function for evaluating flow
thermal_stream = ctr.stream(['x','y','z'], ['theta_s', 'theta_f', 'u', 'v', 'w', 'p'])

# function for evaluating SDF
input_params = {'x': None,
                'y': None,
                'z': None}
limerock_sdf_fn = np_lambdify(limerock.geo_solid.sdf, input_params)

# make plot data
nx, ny, nz = 128, 128, 512
ncells = nx * ny * nz
origin = [x[0] for x in limerock.geo_hr_bounds.values()]
spacing = [(x[1] - x[0])/y for x, y in zip(limerock.geo_hr_bounds.values(), [nx, ny, nz])]
mesh_x, mesh_y, mesh_z = np.meshgrid(np.linspace(origin[0], nx*spacing[0]+origin[0], nx),
                                     np.linspace(origin[1], ny*spacing[1]+origin[1], ny),
                                     np.linspace(origin[2], nz*spacing[2]+origin[2], nz),
                                     indexing='ij')
mesh_x = np.expand_dims(mesh_x, axis=-1)
mesh_y = np.expand_dims(mesh_y, axis=-1)
mesh_z = np.expand_dims(mesh_z, axis=-1)

# evaluate SDF and flow on points
limerock_sdf = {'sdf': limerock_sdf_fn(x=mesh_x, y=mesh_y, z=mesh_z)}
limerock_thermal = thermal_stream({'x': mesh_x, 'y': mesh_y, 'z': mesh_z})
limerock_data = {**limerock_sdf, **limerock_thermal}
cell_data = {}
for key, value in limerock_data.items():
  if str(key) in ['theta_f', 'u', 'v', 'w', 'p']:
    cell_data[str(key)] = np.ascontiguousarray((value*np.heaviside(-limerock_data['sdf'], 0))[...,0])
  elif str(key) in ['theta_s']:
    cell_data[str(key)] = np.ascontiguousarray((value*np.heaviside(limerock_data['sdf'], 0))[...,0])
  else:
    cell_data[str(key)] = np.ascontiguousarray(value[...,0])

# save everything to vti file
imageToVTK("limerock_evaluation", origin, spacing, cellData=cell_data)
```

Listing 140: Evaluating results on a Mesh grid

18.4 Results and Post-processing

To confirm the accuracy of our model, we compare the SimNet results for pressure drop and peak temperature with the OpenFOAM and a commercial solver results, reported in Table 14. The results show good accuracy achieved by the hFTB method. Table 14 demonstrates the impact of mesh refinement on the solution of the commercial solver where with increasing mesh density and mesh quality, the commercial solver results show convergence towards the SimNet results. We show a visualization of the heat sink temperature profile in Figure 80.

Table 13: A comparison for the solver and SimNet results for NVSwitch pressure drop and peak temperature.

Property	OpenFOAM	Commercial Solver	SimNet
Pressure Drop (Pa)	133.96	137.50	150.25
Peak Temperature $(^{\circ}C)$	93.41	95.10	97.35

Table 14: Commercial solver mesh refinement results for NVSwitch pressure drop and peak temperature.

Number of elements	Commercial solver pressure drop (Pa)	SimNet pressure drop (Pa)	Absolute difference (%)	Commercial solver peak temperature $(^{\circ}C)$	SimNet peak temperature $(^{\circ}C)$	Absolute difference (%)
22.4 M	81.27	150.25	84.88	97.40	97.35	0.05
24.7 M	111.76	150.25	34.44	95.50	97.35	1.94
26.9 M	122.90	150.25	22.25	95.10	97.35	2.36
30.0 M	132.80	150.25	13.14	-	-	-
32.0 M	137.50	150.25	9.27	-	-	-

18.5 gPC-Based Surrogate Modeling Accelerated via Transfer Learning

Previously in Chapter 16 we showed that by parameterizing the input of our neural network, we can solve for multiple design parameters in a single run and use that parameterized network for design optimization. Here in this section, we introduce another approach for parameterization and design optimization, which is based on constructing a surrogate using the solution obtained from a limited number of non-parameterized neural network models. Compared to the parameterized network approach that is limited to the CSG module, this approach can be used for parameterization of both constructive solid and STL geometries, and additionally, can offer improved accuracy specially for cases with a high-dimensional parameter space and also in cases where some or all of the design parameters are discrete. However, this approach requires training of multiple neural networks and may require multi-node resources.

We specifically focus on surrogates based on the generalized Polynomial Chaos (gPC) expansions. The gPC is an efficient tool for uncertainty quantification using limited data, and in introduced in Section 1.8. We start off by generating the required number of realizations form the parameter space using a low-discrepancy sequence such as Halton or Sobol. Next, for each realization, we train a separate neural network model. Note that these trainings are independent from each other and therefore, this training step is embarrassingly parallel and can be done on multiple GPUs or nodes. Finally, we train a gPC surrogate that maps the parameter space to the quantities of interest (e.g., pressure drop and peak temperature in the heat sink design optimization problem).

In order to reduce the computational cost of this approach associated with training of multiple models, we use transfer learning, that is, once a model is fully trained for a single realization, it is used for initialization of the other models, and this can significantly reduce the total time to convergence. Transfer learning has been previously introduced in Chapter 14.

Here, to illustrate the gPC surrogate modeling accelerated via transfer learning, we consider the NVIDIA's NVSwitch heat sink introduced in the previous section, and limit the geometry parameters to four fin cut angle parameters, as shown in Figure 81. We then construct a pressure drop surrogate. Similarly, one can also construct a surrogate for the peak temperature and use these two surrogates for design optimization of this heat sink.

The scripts for this example are in the `limerock/limerock_4params_pce_surrogate` directory. Following Section 1.8, we generate 30 geometry realizations according to a Halton sequence by running `sample_generator.py`, as follows

```
# import libraries
import numpy as np
import chaospy

# define parameter ranges
fin_front_top_cut_angle_ranges = (0., np.pi/6.)
fin_front_bottom_cut_angle_ranges = (0., np.pi/6.)
fin_back_top_cut_angle_ranges = (0., np.pi/6.)
fin_back_bottom_cut_angle_ranges = (0., np.pi/6.)

# generate samples
samples = chaospy.generate_samples(order=30,
                                   domain=np.array([fin_front_top_cut_angle_ranges,
                                                    fin_front_bottom_cut_angle_ranges,
```

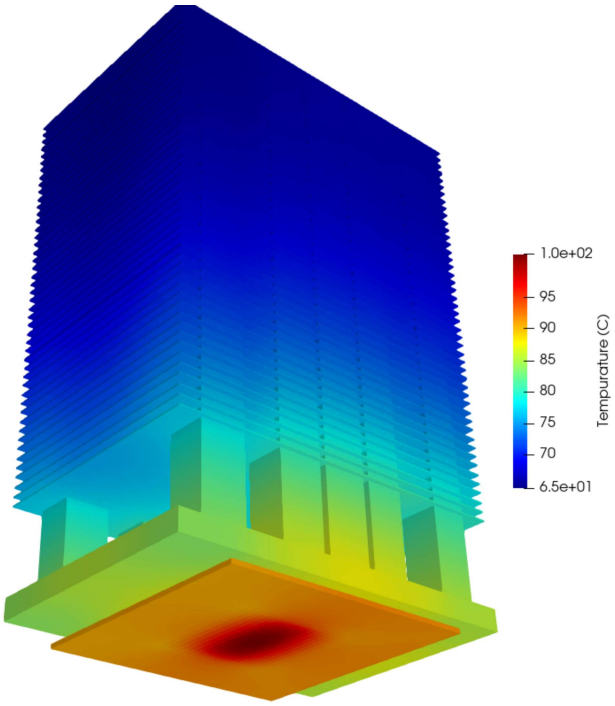


Figure 80: NVSwitch Solid Temperature

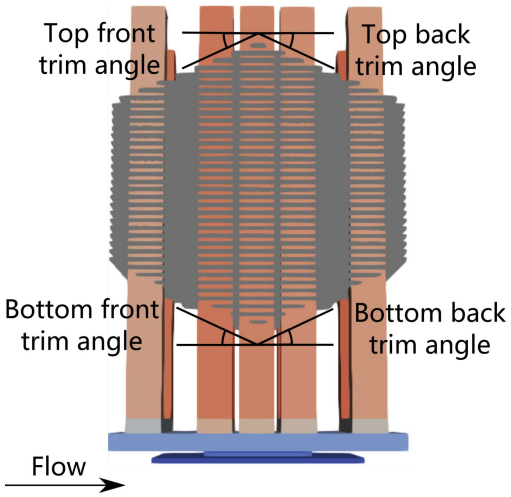


Figure 81: NVSwitch heat sink geomtry parameterization. Each parameter ranges between 0 and $\pi/6$.

```
                                                    fin_back_top_cut_angle_ranges,
                                                    fin_back_bottom_cut_angle_ranges]).T,
                                        rule="halton")
samples = samples.T
np.random.shuffle(samples)
np.savetxt('samples.txt', samples)
```

Listing 141: Generating geometry realizations using a Halton sequence

We then train a separate flow network for each of these realizations using transfer learning. To do this, we change the `sample_id` variable in `stl_to_geo.py`, and then run `limerock_flow_solver.py`. This is repeated until all of the geometry realizations are covered. These flow models are initialized using the trained network for the base geometry (as shown in Figure 79), and are trained for a fraction of the total training iterations for the base geometry, with a smaller learning rate and a faster learning rate decay. This is because we only need to fine-tune these models as opposed to training them from the scratch. Please note that, before we launch the transfer learning runs, a flow network for the base geometry needs to be fully trained.

```
@classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'initialize_network_dir': './network_checkpoint_limerock_fluid_flow_real',  # network dir from a full
     run
        'network_dir': './network_checkpoint_limerock_fluid_flow_real_TL_' + str(sample_id),
        'rec_results_cpu': True,
        'max_steps': 200000,
        'decay_steps': 2500,
        'start_lr': 2.5e-4,
        # 'max_steps': 1000000, # if full run
        # 'decay_steps': 10000, # if full run
        # 'start_lr': 5e-4,  # if full run
        })
```

Listing 142: Training configs for a transfer learning model

Figure 82 shows the front and back pressure results for different runs. It is evident that the pressure has converged faster in the transfer learning runs compared to the base geometry full run, and that transfer learning has reduced the total time to convergence by a factor of 5.

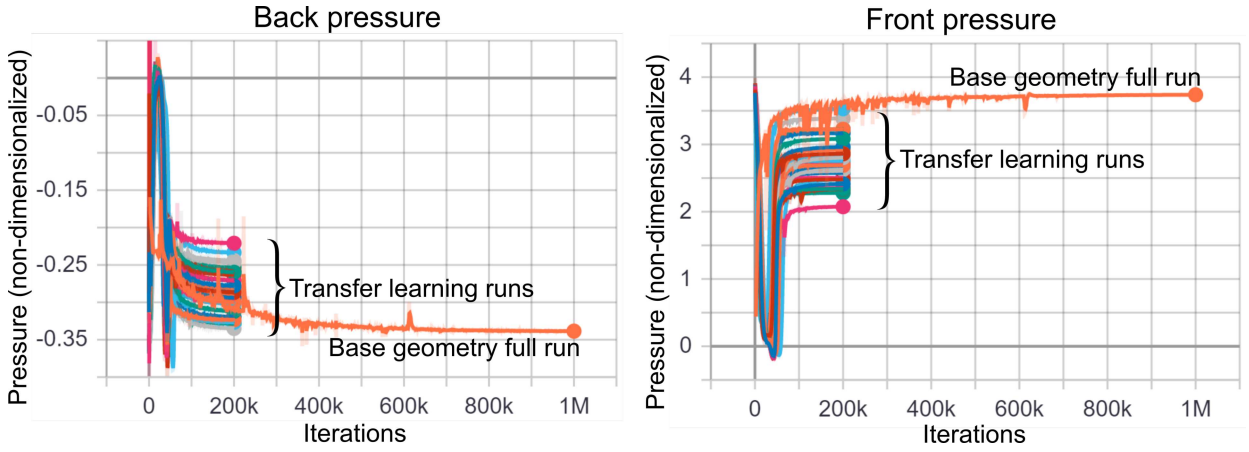


Figure 82: NVSwitch front and back pressure convergence results for different geometries using transfer learning.

Finally, we randomly divide the pressure drop data obtained from these models into training and test sets, and construct a gPC surrogate by running `limerock_pressure_drop_surrogate.py`, as follows:

```
# import libraries
import numpy as np
import csv
import chaospy

# load data
samples = np.loadtxt('samples.txt')
num_samples = len(samples)
```

```
y_vec = []
for i in range(num_samples):
  front_pressure_dir = './checkpoints_flow/network_checkpoint_TL_' + str(i) +'/monitor_domain/results/
    FrontPressure.csv'
  back_pressure_dir = './checkpoints_flow/network_checkpoint_TL_' + str(i) +'/monitor_domain/results/
    BackPressure.csv'
  with open(front_pressure_dir, "r", encoding="utf-8", errors="ignore") as scraped:
    front_pressure = float(scraped.readlines()[-1].split(',')[0])
  with open(back_pressure_dir, "r", encoding="utf-8", errors="ignore") as scraped:
    back_pressure = float(scraped.readlines()[-1].split(',')[0])
  pressure_drop = front_pressure - back_pressure
  y_vec.append(pressure_drop)
y_vec = np.array(y_vec)

# Split data into training and validation
val_portion = 0.16
val_idx = np.random.choice(np.arange(num_samples, dtype=int), int(val_portion*num_samples), replace=False)
val_x, val_y = samples[val_idx], y_vec[val_idx]
train_x, train_y = np.delete(samples, val_idx, axis=0).T, np.delete(y_vec, val_idx).reshape(-1,1)

# Construct the PCE
distribution = chaospy.J(chaospy.Uniform(0., np.pi/6),
                         chaospy.Uniform(0., np.pi/6),
                         chaospy.Uniform(0., np.pi/6),
                         chaospy.Uniform(0., np.pi/6))
expansion = chaospy.generate_expansion(2, distribution)
poly = chaospy.fit_regression(expansion, train_x, train_y)

# PCE closed form
print('__________')
print('PCE closd form:')
print(poly)
print('__________')

# Validation
print('PCE evaluatins:')
for i in range(len(val_x)):
  pred = poly(val_x[i,0],
              val_x[i,1],
              val_x[i,2],
              val_x[i,3])[0]
  print('Sample:', val_x[i])
  print('True val:', val_y[i])
  print('Predicted val:', pred)
  print('Relative error (%):', abs(pred-val_y[i])/val_y[i] * 100)
print('__________')
```

Listing 143: Constructing a gPC surrogate for the pressure drop for a 4-parameter NVSwitch heat sink

Figure 83 shows the gPC surrogate performance on the test set. The relative errors are below 1%, showing the good accuracy of the constructed gPC pressure drop surrogate.

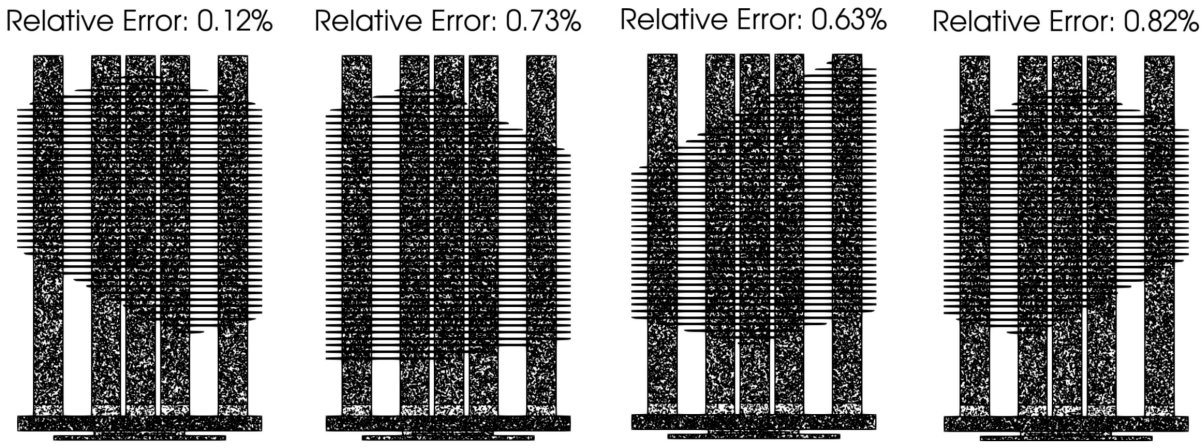


Figure 83: The gPC pressure drop surrogate accuracy tested on four geometries.

19 Case Study: Performance Upgrades and Parallel Processing using Multi-GPU Configurations

19.1 Introduction

In this example we show how to increase the performance of SimNet runs using the advanced features like XLA, TF32, multi-GPU, etc. We will also present some studies that show the scalability of SimNet across multiple GPUs.

19.2 Running jobs using Accelerated Linear Algebra (XLA)

XLA is a domain specific compiler that allows for just-in-time compilation of TensorFlow graphs. PINNs used in SimNet have many peculiarities including the presence of many pointwise operations. Such operations, while being computationally inexpensive, put a large pressure on the memory subsystem of a GPU. XLA allows for kernel fusion, so that many of these operations can be computed simultaneously in a single kernel and thereby reducing the number of memory transfers from GPU memory to the compute units. Kernel fusion using XLA accelerates a single training iteration in SimNet by up to 1.5x.

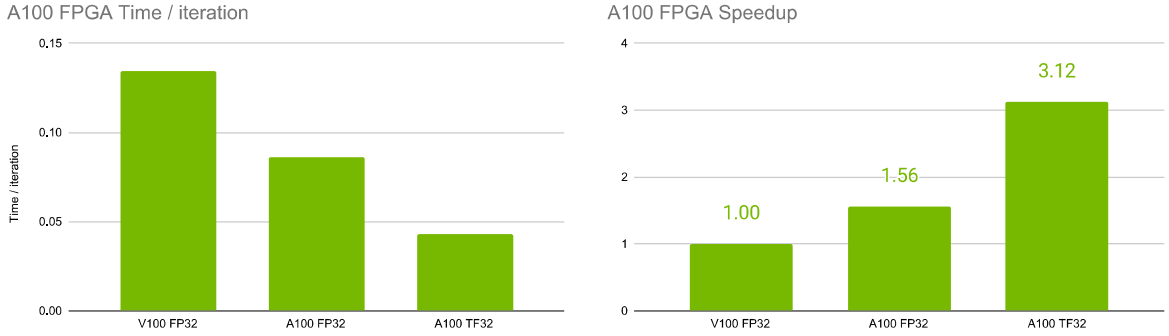
To run a job using XLA, you can pass the `--xla=True` flag while executing the script. This flag can be used for single GPU and multi-GPU runs alike. The following command shows how to run a job using XLA.

```
python fpga_flow_solver.py --xla=True
```

19.3 Running jobs using TF32 math mode

TensorFloat-32 (TF32) is a new math mode available on NVIDIA A100 GPUs for handing matrix math and tensor operations used during the training of a neural network. More details about this feature can be found at https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/.

On A100 GPUs, the TF32 feature is "ON" by default and we do not need to make any modifications to the regular scripts to use this feature. With this feature, we can obtain up to 1.6x speed-up over FP32 on A100 GPUs and up to 3.1x speed-up over FP32 on V100 GPUs for the FPGA problem. This allows us to achieve same results at a reduced training time as shown in Table 15 and loss convergence plots in Figure 85.



(a) Time per iteration

(b) Speed-up

Figure 84: Accelerated training using TF32 on A100 GPUs

19.4 Running jobs using multiple GPUs

To boost performance and to run larger problems, SimNet supports multi-GPU and multi-node scaling using Horovod. This allows for multiple processes, each targeting a single GPU, to perform independent forward and backward passes and aggregate the gradients collectively before updating the model weights. The Figure 86 shows the scaling performance of SimNet on a annular ring test problem (script can be found at `examples/annular_ring/annular_ring.py`) up to 128 V100 GPUs on 16 nodes. The scaling efficiency from 1 to 32 GPUs is more than 95%.

This data parallel fashion of multi-GPU training keeps the number of points sampled per GPU constant while increasing the total effective batch size. We can use this to our advantage to increase the number of points sampled by increasing the number of GPUs allowing us to handle much larger problems.

Table 15: Comparison of results with and without TF32 math mode

Case Description	P_{drop} (Pa)	Compute Time (hrs)
SimNet: Fully Connected Networks with FP32	29.24	86.9
SimNet: Fully Connected Networks with TF32	29.13	39.5
OpenFOAM Solver	28.03	NA
Commercial Solver	28.38	NA

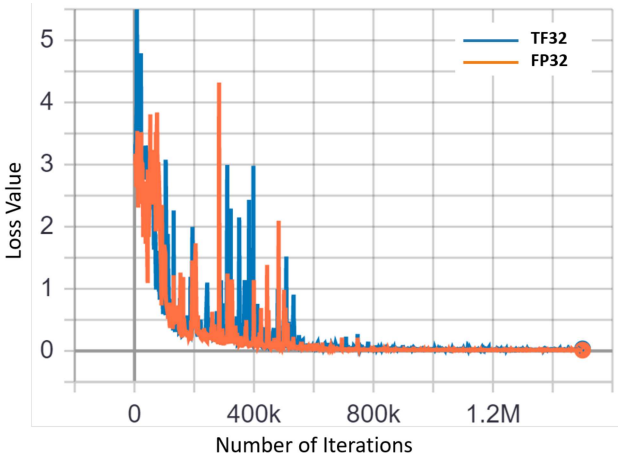


Figure 85: Loss convergence plot for FPGA simulation with TF32 feature

To run a SimNet solution using multiple GPUs on a single compute node, one can first find out the available GPUs using

```
nvidia-smi
```

Once you have found out the available GPUs, you can run the job using `horovodrun -np #GPUs`. Below command shows how to run the job using 2 GPUs.

```
horovodrun -np 2 python fpga_flow_solver.py
```

You can find more information at: https://docs.nvidia.com/deeplearning/frameworks/tensorflow-release-notes/rel_20-07.html#rel_20-07

19.4.1 Automatically increase the learning rate with number of GPUs

In Section 1.6.2.3 of tutorial 1, we describe how the learning rate can be scaled with the batch size to achieve faster time to convergence. As suggested earlier, we can run the same scripts using multi-GPU to achieve larger batch sizes. Doing so, the time per iteration remains fairly constant and the benefits are mostly in terms of large problem solution (Figure 86). However, we can also decrease the total time to convergence by scaling the learning rate linearly with the number of GPUs. As described in [16], simply increasing the learning rate can cause the model to diverge at large batch sizes. This can be fixed by using an initial learning rate warm-up.

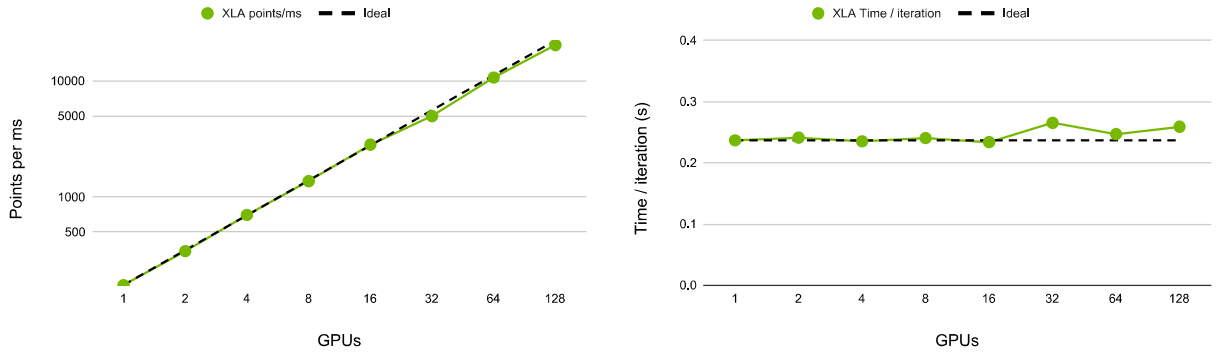
Below are the code changes are required to the script to make use of this feature (with a gradual warm-up) in SimNet.

```
...
from simnet.learning_rate import ExponentialDecayLRWithWarmup

class FPGAFlowSolver(Solver):
  train_domain = FPGAFlowTrain
  monitor_domain = FPGAFlowMonitor
  val_domain = FPGAFlowVal
```

```
  arch = FourierNetArch
  lr = ExponentialDecayLRWithWarmup

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

    self.frequencies = ('axis,diagonal', [i for i in range(35)])

    self.equations = (NavierStokes(nu=nu, rho=rho, dim=3, time=False).make_node(stop_gradients=['nu', 'nu__x', '
     nu__y', 'nu__z'])
                      + IntegralContinuity(dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    self.nets = [flow_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_fpga_fluid_flow',
        'rec_results_cpu': True,
        'max_steps': 4000000,
        'decay_steps': 15000,
        'warmup_type': 'gradual',
        'warmup_steps': 30000,
        'rec_results_freq': 5000,
        })

if __name__ == '__main__':
  ctr = SimNetController(FPGAFlowSolver)
  ctr.run()
```

Listing 144: Exponential learning rate decay with warm-up for multi-GPU runs

Figure 86 shows the weak scaling performance of SimNet for the annular ring problem. SimNet scales very well up to 128 GPUs with an almost constant time per iteration. When coupled with the learning rate scaling with warmup as described in Section 1.6.2.3, this can lead to accelerated time to convergence. Figure 87 shows the learning rate schedule and the loss function evolution as the number of GPUs is increased from 1 to 16 for the NVSwitch heatsink case. For the multi-GPU cases, the learning rate is gradually increased from the baseline case and this allows the model to train without diverging early on and allows the model to converge faster as a result of the increased global batch size coupled with the increased learning rate.



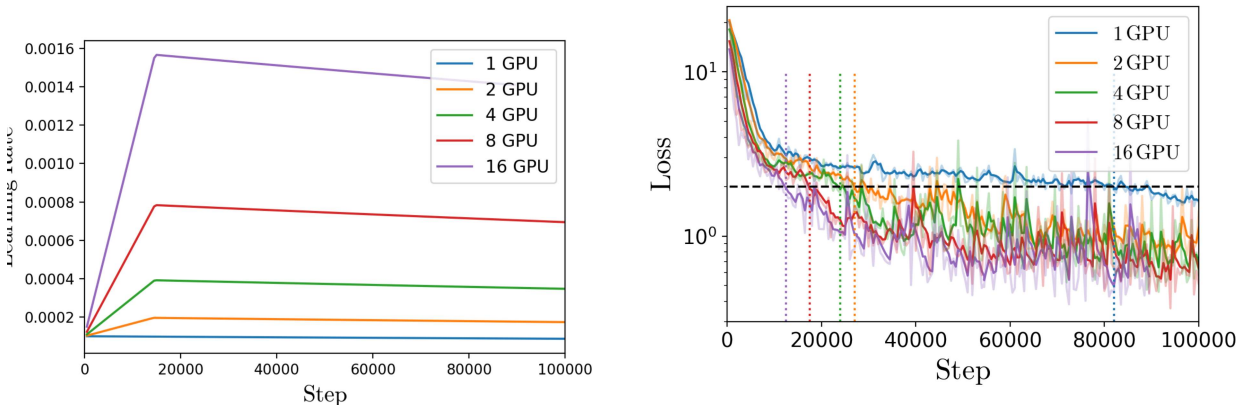
(a) Points per ms

(b) Time per iteration

Figure 86: SimNet weak scaling plots for flow over an annular ring problem

19.4.2 Strong scaling to multiple GPUs for faster time to convergence

Weak scaling with learning rate schedules is an efficient way to get accelerated convergence. But in some cases, large batch training issues might reduce the effectiveness of this approach. Strong scaling the problem is a viable solution in these cases. With strong scaling, a fixed batch size is split across multiple GPUs in order to reduce the wall clock computation time. As the number of GPUs is increased, the global batch size remains constant and the batch size per GPU reduces proportionally. Figure 88 shows the time per iteration for the 3D Taylor-Green vortex problem as the problem is strong scaled. For this problem, we are able to scale the problem efficiently up to 128 GPUs. We get a

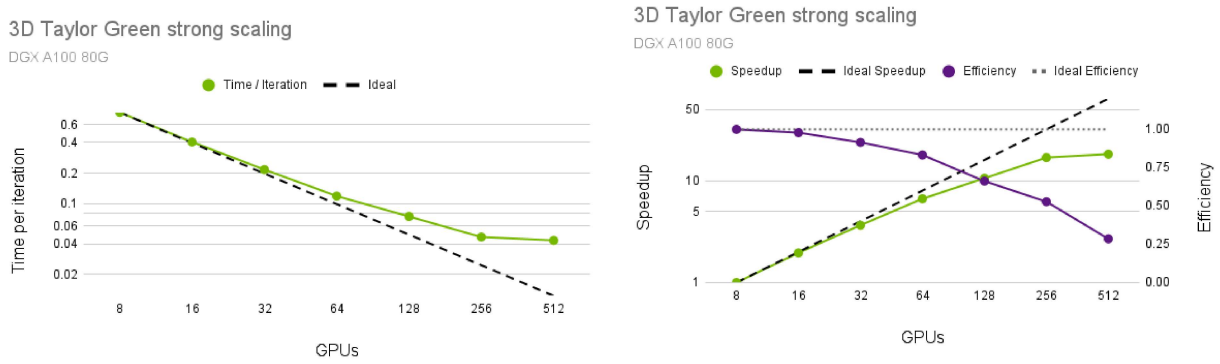


(a) Learning rate

(b) Loss function value

Figure 87: Accelerating time to convergence using multiple GPUs and learning rate schedules.

further speed-up going to 512 GPUs but with a loss in efficiency due to an increase in communication time while the computation time decreases.



(a) Time per iteration

(b) Speedup (green, left axis) and scaling efficiency (purple, right axis)

Figure 88: Strong scaling results for the Taylor-Green vortex problem on A100 GPUs

References

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[2] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017.

[3] Luning Sun, Han Gao, Shaowu Pan, and Jian-Xun Wang. Surrogate modeling for fluid flows based on physics-constrained deep learning without simulation data. *Computer Methods in Applied Mechanics and Engineering*, 361:112732, 2020.

[4] DL Young, CH Tsai, and CS Wu. A novel vector potential formulation of 3d navier–stokes equations with through-flow boundaries by a local meshless method. *Journal of Computational Physics*, 300:219–240, 2015.

[5] Mohammad Amin Nabian, Rini Jasmine Gladstone, and Hadi Meidani. Efficient training of physics-informed neural networks via importance sampling. *arXiv preprint arXiv:2104.12325*, 2021.

[6] John H Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2(1):84–90, 1960.

[7] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. In *International Conference on Machine Learning*, pages 5301–5310, 2019.

[8] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *arXiv preprint arXiv:2003.08934*, 2020.

[9] Matthew Tancik, Pratul P Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan T Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *arXiv preprint arXiv:2006.10739*, 2020.

[10] Sifan Wang, Yujun Teng, and Paris Perdikaris. Understanding and mitigating gradient pathologies in physics-informed neural networks. *arXiv preprint arXiv:2001.04536*, 2020.

[11] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385, 2015.

[12] Vincent Sitzmann, Julien NP Martel, Alexander W Bergman, David B Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *arXiv preprint arXiv:2006.09661*, 2020.

[13] Justin Sirignano and Konstantinos Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of computational physics*, 375:1339–1364, 2018.

[14] Rizal Fathony, Anit Kumar Sahu, Devin Willmott, and J Zico Kolter. Multiplicative filter networks. 2021.

[15] Alex Kendall, Yarin Gal, and Roberto Cipolla. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7482–7491, 2018.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[17] Ameya D Jagtap, Kenji Kawaguchi, and George Em Karniadakis. Adaptive activation functions accelerate convergence in deep and physics-informed neural networks. *Journal of Computational Physics*, 404:109136, 2020.

[18] Dietrich Braess. *Finite elements: Theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, 2007.

[19] David Gilbarg and Neil S Trudinger. *Elliptic partial differential equations of second order*. springer, 2015.

[20] Lawrence C Evans. Partial differential equations and monge-kantorovich mass transfer. *Current developments in mathematics*, 1997(1):65–126, 1997.

[21] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. Variational physics-informed neural networks for solving partial differential equations. *arXiv preprint arXiv:1912.00873*, 2019.

[22] Ehsan Kharazmi, Zhongqiang Zhang, and George Em Karniadakis. hp-vpinns: Variational physics-informed neural networks with domain decomposition. *arXiv preprint arXiv:2003.05385*, 2020.

[23] Dongbin Xiu. *Numerical methods for stochastic computations: a spectral method approach*. Princeton university press, 2010.