9 Interface problem by Variational method

9.1 Introduction

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

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

Prerequisites

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

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

```
pip install quadpy
```

9.2 Problem Description

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

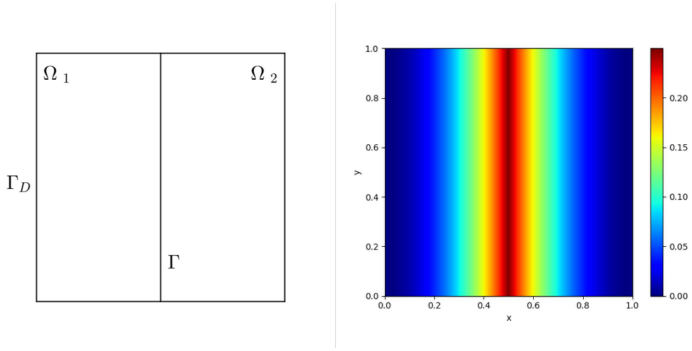


Figure 39: Left: Domain of interface problem. Right: True Solution

The PDEs for the problem are defined as

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

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

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

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

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

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

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

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

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

9.3 Variational Form

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

Variational form for Continuous type formulation :

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

Variational form for Discontinuous type formulation :

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

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

9.4 Continuous type formulation

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

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

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

Listing 42: Importing required packages

9.4.1 Creating the Geometry

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

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

Listing 43: Defining domain

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

9.4.2 Defining the Boundary conditions and Equations to solve

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

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

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

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

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

Listing 44: Defining TrainDomain

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

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

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

9.4.3 Creating the Validation Domains

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

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

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

Listing 45: Defining ValidationDomain and InferenceDomain

9.4.4 Creating the Variational Loss and Solver

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 49: Making test functions

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

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

Listing 50: Evaluating test functions

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

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

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

Listing 51: Forming integrals

The whole code to make the Solver is the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Listing 52: Code for Solver

9.4.5 Results and Post-processing

The results for the problem are shown in Figure 40.

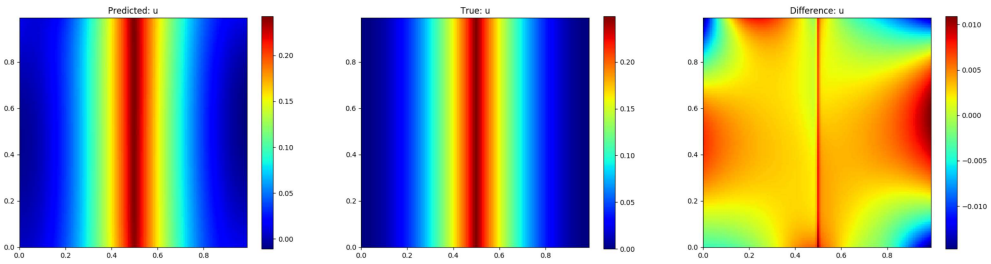


Figure 40: Left: SimNet. Center: Analytical. Right: Difference.

9.5 Discontinuous type formulation

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

9.5.1 Defining the Boundary conditions and Equations

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

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

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

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

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

Listing 53: `TrainDomain`

9.5.2 Creating the Variational Loss and Solver

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

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

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

Listing 54: Making the PINNs' PDEs constraints and continuity

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

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

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

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

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

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

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

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

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

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

Listing 55: Making the variational loss for discontinuous formulation

9.5.3 Results and Post-processing

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

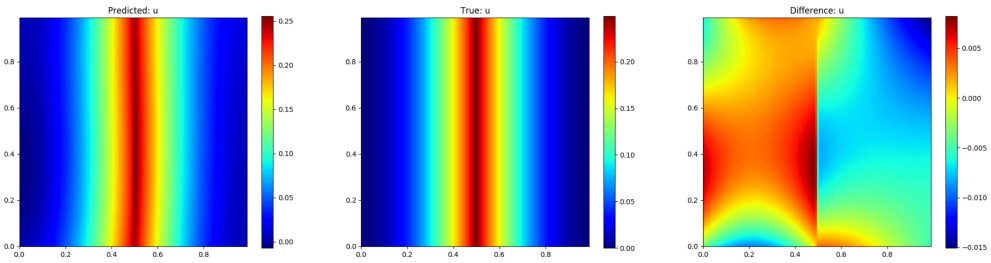


Figure 41: Left: SimNet. Center: Analytical. Right: Difference.

9.6 Quadrature

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

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

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

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

Listing 56: Quadrature rule for Ω

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

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

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

Listing 57: Quadrature rule for Γ

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

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

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

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

Listing 58: `TrainDomain` with quadrature rule

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

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

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

Listing 59: Fetching the quadrature points and weights

And then, the integrals can be formed by

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

Listing 60: Forming the integrals

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

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

9.7 Point source and Dirac Delta function

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

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

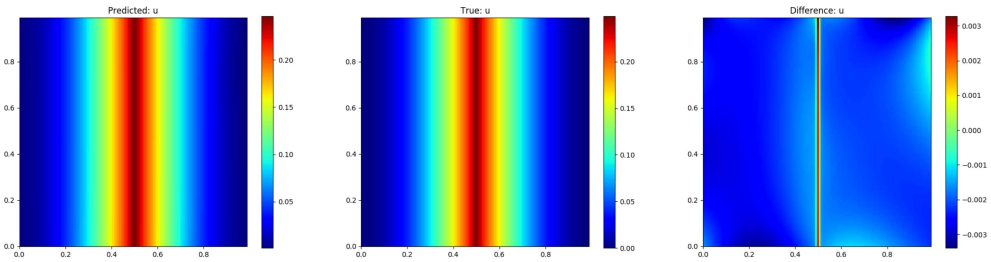


Figure 42: Left: SimNet. Center: Analytical. Right: Difference.

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

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

for all continuous compactly supported functions f.

In this subsection, we solve the following problem:

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

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

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

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

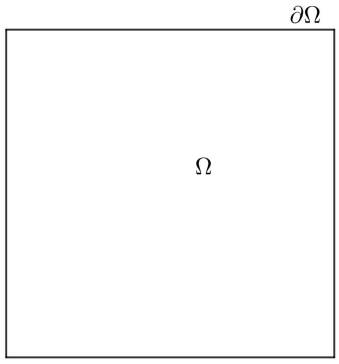


Figure 43: Domain for the point source problem.

9.7.1 Creating the Geometry

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

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

Listing 61: Making the geometry

9.7.2 Creating the Variational Loss and Solver

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

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

Listing 62: Making the right hand side term

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

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

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

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

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

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

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

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


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

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

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


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

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

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

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

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

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

Listing 63: Making the Solver

9.7.3 Results and Post-processing

The results for the problem are shown in Figure 44.

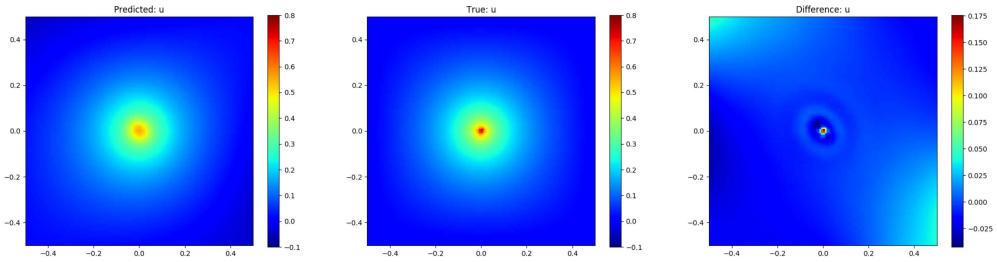


Figure 44: Left: SimNet. Center: Analytical. Right: Difference.

10 Electromagnetics: Frequency Domain Maxwell's Equation

10.1 Introduction

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

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

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

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

Prerequisites

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

10.2 Problem 1: 2D Waveguide Cavity

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

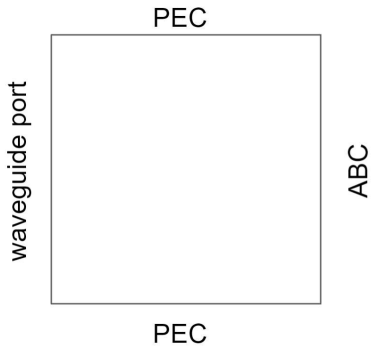


Figure 45: Domain of 2D waveguide

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

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

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

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

10.2.1 Case Setup

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

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

Listing 64: Import libraries for EM simulation

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

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

Listing 65: Define `sympy` variables and geometry.

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

```
eigenmode = [1]
wave_number = 32.    # wave_number = freq/c
waveguide_port = Number(0)
for k in eigenmode:
  waveguide_port += sin(k*pi*y/height)
```

Listing 66: Define the waveguide port

To validate the result, we import the `csv` files for the validation domain below.

```
# validation data
mapping = {'x': 'x', 'y': 'y', 'u': 'u'}
validation_var = csv_to_dict('validation/2Dwaveguide_16_1.csv', mapping)
validation_invar_numpy = {key: value for key, value in validation_var.items() if key in ['x', 'y']}
validation_outvar_numpy = {key: value for key, value in validation_var.items() if key in ['u']}
```

Listing 67: Import walidation data

Now, we can define the `TrainDomain`. The BCs are defined based on the explanations provided above. Please note that in the interior domain, the weights of the PDE is `1.0/wave_number**2`. This is because when wavenumber is large, the PDE loss will be very large in the beginning and will potentially break the training. We have found that by this weighting method we can eliminate this phenomenon.

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

    #walls
    Wall = rec.boundary_bc(outvar_sympy={'u': 0.},
                           criteria=Or(Eq(y, 0), Eq(y, height)),
                           lambda_sympy={'lambda_u': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="PEC")

    Wall = rec.boundary_bc(outvar_sympy={'u': waveguide_port},
                           criteria=Eq(x, 0),
                           lambda_sympy={'lambda_u': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="Waveguide_port")

    Wall = rec.boundary_bc(outvar_sympy={'normal_gradient_u': 0.},
```

```
                            criteria=Eq(x, width),
                            lambda_sympy={'lambda_normal_gradient_u': 10.},
                            fixed_var=False,
                            batch_size_per_area=100)
    self.add(Wall, name="ABC")

    # interior
    interior = rec.interior_bc(outvar_sympy={'helmholtz': 0.},
                               bounds={x: (0, width), y: (0, height)},
                               lambda_sympy={'lambda_helmholtz': 1.0/wave_number**2},
                               fixed_var=False,
                               batch_size_per_area=1000)
    self.add(interior, name="Interior")
```

Listing 68: Training domain for the 2D Waveguide Cavity

Validation domain has been implemented to verify the results.

```
class WaveguideVal(ValidationDomain):
  def __init__(self, **config):
    super(WaveguideVal, self).__init__()
    val = Validation.from_numpy(validation_invar_numpy, validation_outvar_numpy)
    self.add(val, name='Val')
```

Listing 69: Validation domain for the 2D Waveguide Cavity

Inference domain has been implemented to plot the results.

```
class WaveguideInference(InferenceDomain):
  def __init__(self, **config):
    super(WaveguideInference, self).__init__()
    interior_points = rec.sample_interior(100000, bounds={x: (0, width), y: (0, height)})
    inf = Inference(interior_points, ['u'])
    self.add(inf, name='Inf'+str(int(wave_number)))
```

Listing 70: Inference domain for the 2D Waveguide Cavity

For wave simulation, since the result is always periodic, Fourier feature will greatly helpful for the convergence and accuracy. The frequency of the Fourier feature can be implied by the wavenumber. The below block of code shows the solver setup.

```
class WaveguideSolver(Solver):
    train_domain = HemholtzTrain
    inference_domain = WaveguideInference
    val_domain = WaveguideVal
    arch = ModifiedFourierNetArch

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

    self.equations = (HelmholtzEquation(u='u', k=wave_number, dim=2).make_node()
                      + GradNormal(T='u', dim=2, time=False).make_node())

    self.arch.frequencies = ('axis,diagonal', [i / 2. for i in range(int(wave_number)*2+1)])
    self.arch.frequencies_params = ('axis,diagonal', [i / 2. for i in range(int(wave_number)*2+1)])

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

    @classmethod
    def update_defaults(cls, defaults):
      defaults.update({
          'network_dir': './network_checkpoint_waveguide2D_TMz',
          'start_lr': 1e-3,
          'decay_steps': 10000,
          'max_steps': 500000,
          'layer_size': 512,
          'nr_layers': 6,
          'xla': True,
          })

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

Listing 71: Solver for the 2D Waveguide Cavity

10.3 Problem 2: 2D Dielectric slab waveguide

In this section, we do the 2D waveguide simulation with a dielectric slab. The problem setup is almost same as before except there is a horizontal dielectric slab in the middle of the domain. The domain is shown in Figure 46. In the

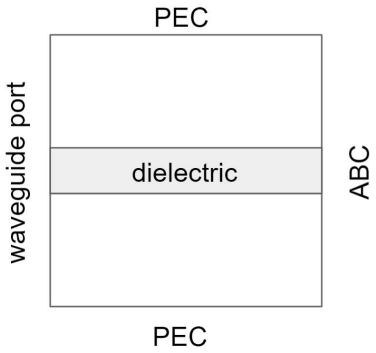


Figure 46: Domain of 2D Dielectric slab waveguide

dielectric, we set the relative permittivity $\epsilon_r = 2$. That is,

$$\epsilon_r = \begin{cases} 2 & \text{in dielectric slab,} \\ 1 & \text{otherwise.} \end{cases}$$

All the other settings are kept the same to the previous example.

10.3.1 Case setup

For the sake of simplicity, we only show the parts of code that are different than the previous example. The main difference is the spatially dependent permittivity. First, we will need to compute the eigenfunctions on the left boundary.

```
# helper function for computing laplacian eigen values
def Laplacian_1D_eig(a, b, N, eps=lambda x: np.ones_like(x), k=3):
  n = N - 2
  h = (b-a)/(N-1)

  L = diags([1, -2, 1], [-1, 0, 1], shape=(n, n))
  L = -L/h**2

  x = np.linspace(a, b, num=N)
  M = diags([eps(x[1:-1])], [0])

  eigvals, eigvecs = eigsh(L, k=k, M=M, which="SM")
  eigvecs = np.vstack((np.zeros((1,k)), eigvecs, np.zeros((1,k))))
  norm_eigvecs = np.linalg.norm(eigvecs, axis=0)
  eigvecs /= norm_eigvecs
  return eigvals.astype(np.float32), eigvecs.astype(np.float32), x.astype(np.float32)
```

Listing 72: 1D eigensolver

For the geometry part, we will need to define the slab and corresponding permittivity function.

```
len_slab = 0.6
eps0 = 1.
eps1 = 2.
eps_numpy = lambda y: np.where(np.logical_and(y>(height-len_slab)/2, y<(height+len_slab)/2), eps1, eps0)
eps_sympy = sqrt(eps0+(Heaviside(y-(height-len_slab)/2)-Heaviside(y-(height+len_slab)/2))*(eps1-eps0))
```

Listing 73: Define the slab and corresponding permittivity function

There is a square root in `eps_sympy` because in the `HelmholtzEquation`, the wavenumber will be squared. Next, based on the permittivity function, we use the eigensolver to get the numerical waveguide port.

```
eigvals, eigvecs, yv = Laplacian_1D_eig(0, height, 1000, eps=eps_numpy, k=3)
yv = yv.reshape((-1, 1))
eigenmode = [1]
wave_number = 16.    # wave_number = freq/c
waveguide_port_invar_numpy = {'x': np.zeros_like(yv), 'y': yv}
waveguide_port_outvar_numpy = {'u': 10*eigvecs[:, 0:1]}
```

Listing 74: Get the waveguide port

Now, we can define the `TrainDomain`. The only difference here is the left boundary, which will be given by a `numpy` array. We show only the modified BC below:

```
Wall = BC.from_numpy(waveguide_port_invar_numpy,
                     waveguide_port_outvar_numpy,
                     batch_size=200,
                     lambda_numpy={'lambda_u': np.full_like(yv, 0.5)})
self.add(Wall, name="Waveguide_port")
```

Listing 75: Training domain for 2D Dielectric slab

In the `Solver`, we set the `k` as the product of wavenumber and permittivity function. Also, we update the frequency for the Fourier features to suit the problem.

```
class WaveguideSolver(Solver):
    train_domain = HemholtzTrain
    inference_domain = WaveguideInference
    arch = ModifiedFourierNetArch

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

        self.equations = (HelmholtzEquation(u='u', k=wave_number*eps_sympy, dim=2).make_node()
                          + GradNormal(T='u', dim=2, time=False).make_node())

        self.arch.frequencies = ('axis,diagonal', [i / 2. for i in range(int(wave_number*np.sqrt(eps1))*2+1)])
        self.arch.frequencies_params = ('axis,diagonal', [i / 2. for i in range(int(wave_number*np.sqrt(eps1))
    *2+1)])
        wave_net = self.arch.make_node(name='wave_net',
                                       inputs=['x', 'y'],
                                       outputs=['u'])
        self.nets = [wave_net]
```

Listing 76: Solver for 2D Dielectric slab

10.3.2 Results

The full code of this example can be found in `examples/waveguide/dielectric_slab_waveguide_2D_TMz.py`. We do the simulation with wavenumber equals 16 and 32, respectively. The results are shown in Figure 47.

10.4 Problem 3: 3D waveguide cavity

In this example, we show how to setup a 3D waveguide simulation in SimNet. Unlike the previous examples, we will use the features in SimNet to define the boundary condition. The geometry is $\Omega = [0, 2]^3$, as shown in Figure 48.

10.4.1 Problem setup

We will solve the 3D frequency domain Maxwell's equation for electronic field $\mathbf{E} = (E_x, E_y, E_z)$:

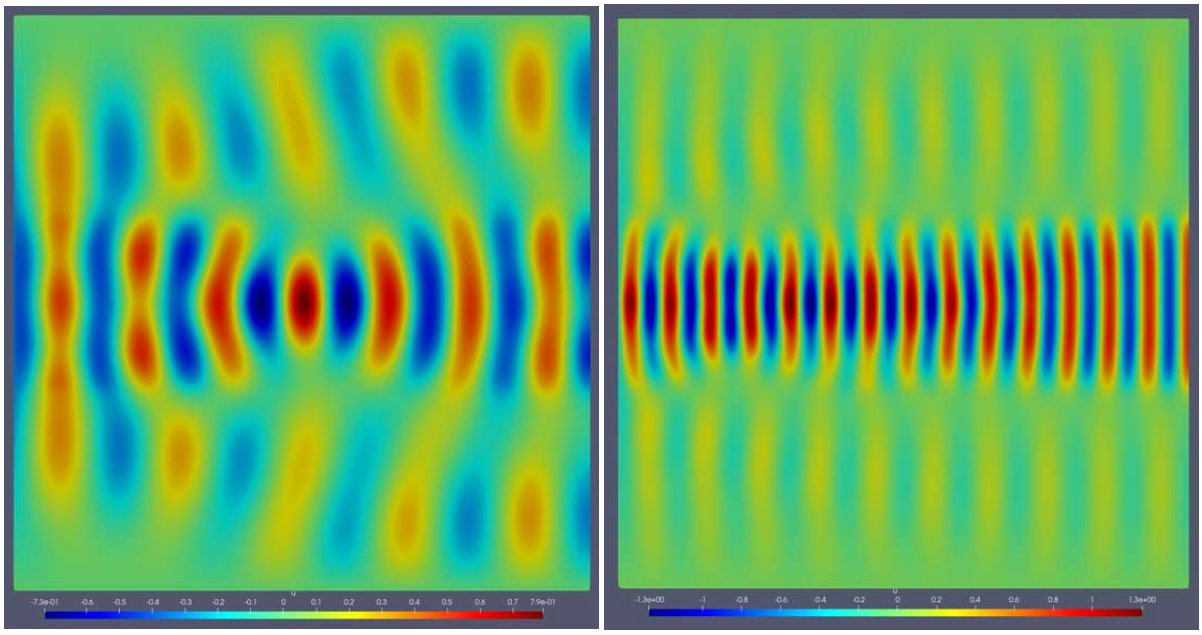
$$\nabla \times \nabla \times \mathbf{E} + \epsilon_r k^2 \mathbf{E} = 0,$$

where ϵ_r is the permittivity, and the k is the wavenumber. Note that, currently SimNet only support real permittivity and wavenumber. For the sake of simplicity, we assume the permeability $\mu_r = 1$. As before, waveguide port has been applied on the left. We apply absorbing boundary condition on the right side and PEC for the rest. In 3D, the absorbing boundary condition for real form reads

$$\mathbf{n} \times \nabla \times \mathbf{E} = 0,$$

while the PEC is

$$\mathbf{n} \times \mathbf{E} = 0.$$



(a) SimNet, wavenumber=16 (b) SimNet, wavenumber=32

Figure 47: Result of simulation. Eigenmode=1

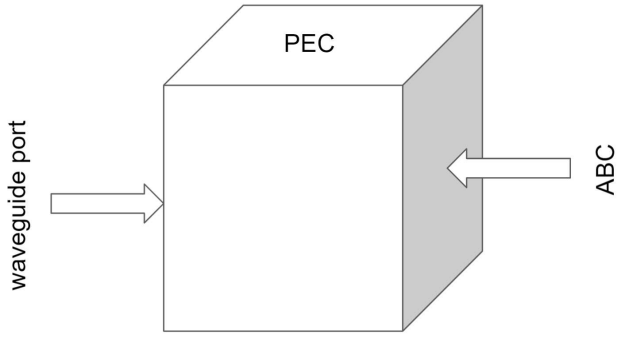


Figure 48: 3D waveguide geometry

10.4.2 Case setup

In this section, we will show how to use SimNet to setup the 3D frequency EM solver, especially for the boundary conditions.

We first import the necessary libraries.

```
from sympy import Symbol, pi, sin, Number, Eq
from sympy.logic.boolalg import And
from simnet.solver import Solver
from simnet.dataset import TrainDomain, InferenceDomain
from simnet.data import Inference
from simnet.sympy_utils.geometry_3d import Box
from simnet.PDES.Electromagnetic import PEC_3D, SommerfeldBC_real_3D, Maxwell_Freq_real_3D
from simnet.controller import SimNetController
from simnet.architecture.modified_fourier_net import ModifiedFourierNetArch
```

Listing 77: Import libraries

And, as before, we define `sympy` variables, geometry and waveguide function.

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

```
# params for domain
length = 2
height = 2
width = 2

eigenmode = [2]
wave_number = 32.    # wave_number = freq/c
waveguide_port = Number(0)
for k in eigenmode:
    waveguide_port += sin(k*pi*y/length)*sin(k*pi*z/height)

# define geometry
rec = Box((0, 0, 0),
          (width, length, height))
```

Listing 78: Preparations for the simulation

Then, we define the `TrainDomain` with PDEs, and all boundary conditions. The 3D Maxwell's equations has been implemented in `Maxwell_Freq_real_3D`, PEC has been implemented in `PEC_3D`, and absorbing boundary condition has been implemented in `SommerfeldBC_real_3D`. We may use these features directly to apply the corresponding constraints.

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

    #walls
    Wall = rec.boundary_bc(outvar_sympy={'PEC_3D_x': 0., 'PEC_3D_y': 0., 'PEC_3D_z': 0.},
                           criteria=And(~Eq(x, 0), ~Eq(x, width)),
                           lambda_sympy={'lambda_PEC_3D_x': 100.,
                                         'lambda_PEC_3D_y': 100.,
                                         'lambda_PEC_3D_z': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="PEC")

    Wall = rec.boundary_bc(outvar_sympy={'uz': waveguide_port},
                           criteria=Eq(x, 0),
                           lambda_sympy={'lambda_uz': 100.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="Waveguide_port")

    Wall = rec.boundary_bc(outvar_sympy={'SommerfeldBC_real_3D_x': 0.,
                                         'SommerfeldBC_real_3D_y': 0.,
                                         'SommerfeldBC_real_3D_z': 0.},
                           criteria=Eq(x, width),
                           lambda_sympy={'lambda_SommerfeldBC_real_3D_x': 10.,
                                         'lambda_SommerfeldBC_real_3D_y': 10.,
                                         'lambda_SommerfeldBC_real_3D_z': 10.},
                           fixed_var=False,
                           batch_size_per_area=100)
    self.add(Wall, name="ABC")

    # interior
    interior = rec.interior_bc(outvar_sympy={'Maxwell_Freq_real_3D_x': 0.,
                                             'Maxwell_Freq_real_3D_y': 0.,
                                             'Maxwell_Freq_real_3D_z': 0.},
                               bounds={x: (0, width), y: (0, length), z: (0, height)},
                               lambda_sympy={'lambda_Maxwell_Freq_real_3D_x': 1.0/wave_number**2,
                                             'lambda_Maxwell_Freq_real_3D_y': 1.0/wave_number**2,
                                             'lambda_Maxwell_Freq_real_3D_z': 1.0/wave_number**2},
                               fixed_var=False,
                               batch_size_per_area=1000)
    self.add(interior, name="Interior")
```

Listing 79: Define the train domain

Note that we are working in 3D, so the PDEs, PEC and absorbing boundaries have three output components.

Inference domain has been defined to check the result.

```
class WaveguideInference(InferenceDomain):
  def __init__(self, **config):
    super(WaveguideInference, self).__init__()
    interior_points = rec.sample_interior(100000, bounds={x: (0, width), y: (0, length), z: (0, height)})
    inf = Inference(interior_points, ['ux', 'uy', 'uz'])
```

```
    self.add(inf, name='Inf'+str(wave_number).zfill(4))
```

Listing 80: Define the inference domain

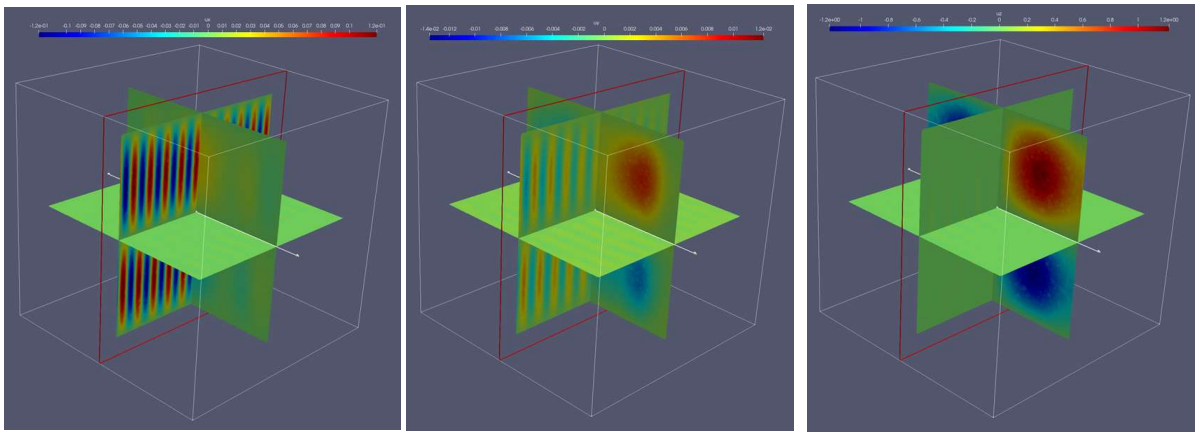
We finalize the code by defining the `Solver` and necessary hyperparameters for the neural network.

```
class WaveguideSolver(Solver):
  train_domain = HemholtzTrain
  inference_domain = WaveguideInference
  arch = ModifiedFourierNetArch

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

    self.equations = (Maxwell_Freq_real_3D(k=wave_number).make_node()
                      + PEC_3D().make_node()
                      + SommerfeldBC_real_3D().make_node())

    self.arch.frequencies = ('axis,diagonal', [i / 2. for i in range(int(wave_number)+1)])
    self.arch.frequencies_params = ('axis,diagonal', [i / 2. for i in range(int(wave_number)+1)])
    wave_net = self.arch.make_node(name='wave_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['ux', 'uy', 'uz'])
    self.nets = [wave_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_waveguide3D',
        'start_lr': 1e-3,
        'decay_steps': 10000,
        'max_steps': 500000,
        'layer_size': 512,
        'nr_layers': 6,
        'xla': True,
        })
```

Listing 81: Define neural network

10.4.3 Results

The full code of this example can be found in `examples/waveguide/waveguide3D.py`. We set the wavenumber equals 32 and use second eigenmode for y and z. The silces of the three components are shown in Figure 49.

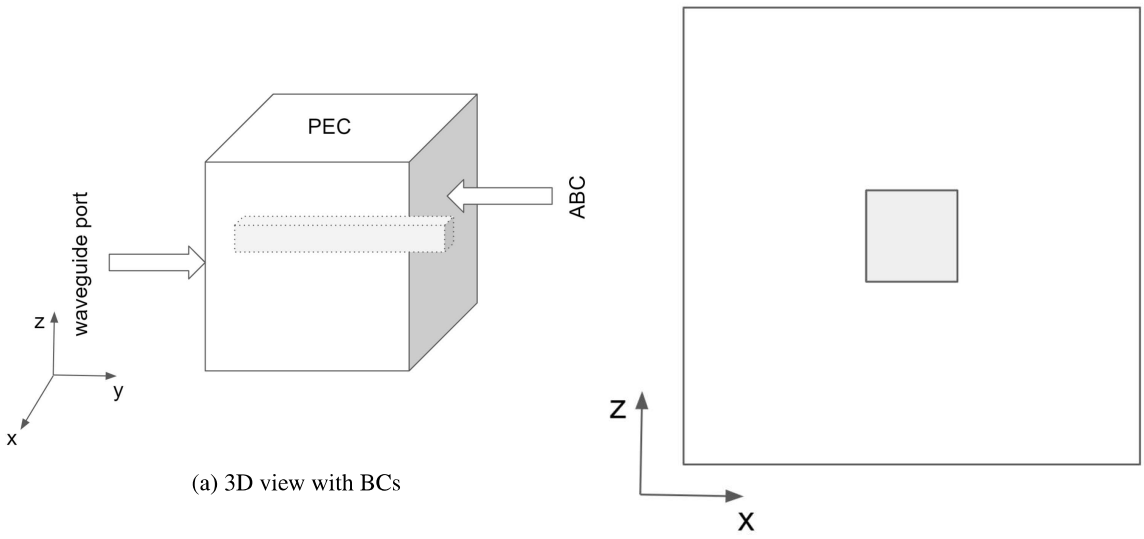


(a) 3D waveguide, E_x (b) 3D waveguide, E_y (c) 3D waveguide, E_z

Figure 49: Results

10.5 Problem 4: 3D Dielectric slab waveguide

In this example, we will show a 3D dielectric slab waveguide. In this case, we consider a unit cude $[0, 1]^3$ with a dielectric slab centered in the middle along y axis. The length of the slab is 0.2. Figure 50 shows the whole geometry and an xz cross-section that shows the dielectric slab.



(a) 3D view with BCs

(b) xz cross-sectional view

Figure 50: Geometry of 3D dielectric slab

The permittivity is defined as follows

$$\epsilon_r = \begin{cases} 1.5 & \text{in dielectric slab,} \\ 1 & \text{otherwise.} \end{cases}$$

10.5.1 Case setup

For the sake of simplicity, we only show the differences compared to the last example. The main difference for this simulation is that we will need to import the waveguide result from a `csv` file and then use that as the waveguide port boundary condition.

Let us first define the geometry and the `sympy` permittivity function:

```
# params for domain
length = 1
height = 1
width = 1

len_slab = 0.2
eps0 = 1.
eps1 = 1.5
eps_sympy = sqrt(eps0+(Heaviside(y+len_slab/2)-Heaviside(y-len_slab/2))
                 *(Heaviside(z+len_slab/2)-Heaviside(z-len_slab/2))*(eps1-eps0))

# define geometry
rec = Box((-width/2, -length/2, -height/2),
          (width/2, length/2, height/2))
```

Listing 82: Define the domain and permittivity function

Note that to define the piece-wise `sympy` functions, we should use `Heaviside` instead of `Piecewise` as the later cannot be complied in SimNet for the time being.

Next, we import the waveguide data.

```
mapping = {'x': 'x', 'y': 'y', **{'u'+str(k): 'u'+str(k) for k in range(6)}}
data_var = csv_to_dict('validation/2Dwaveguideport.csv', mapping)
wave_number = 32.    # wave_number = freq/c
waveguide_port_invar_numpy = {'x': np.zeros_like(data_var['x']), 'y': data_var['x'], 'z': data_var['y']}
waveguide_port_outvar_numpy = {'uz': data_var['u0']}
```

Listing 83: Import waveguide data

In validation/2Dwaveguideport.csv, there are six eigenmodes. You may try different modes to explore more interesting results.

Then, we define the TrainDomain. Please note that we use imported data as the waveguide port boundary condition.

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

        #walls
        Wall = rec.boundary_bc(outvar_sympy={'PEC_3D_x': 0., 'PEC_3D_y': 0., 'PEC_3D_z': 0.},
                               criteria=And(~Eq(x, -width/2), ~Eq(x, width/2)),
                               lambda_sympy={'lambda_PEC_3D_x': 100.,
                                             'lambda_PEC_3D_y': 100.,
                                             'lambda_PEC_3D_z': 100.},
                               fixed_var=False,
                               batch_size_per_area=100)
        self.add(Wall, name="PEC")

        Wall = BC.from_numpy(waveguide_port_invar_numpy,
                             waveguide_port_outvar_numpy,
                             batch_size=200,
                             lambda_numpy={'lambda_uz': np.full_like(waveguide_port_invar_numpy['x'], 0.5)})
        self.add(Wall, name="Waveguide_port")

        Wall = rec.boundary_bc(outvar_sympy={'SommerfeldBC_real_3D_x': 0.,
                                             'SommerfeldBC_real_3D_y': 0.,
                                             'SommerfeldBC_real_3D_z': 0.},
                               criteria=Eq(x, width/2),
                               lambda_sympy={'lambda_SommerfeldBC_real_3D_x': 10.,
                                             'lambda_SommerfeldBC_real_3D_y': 10.,
                                             'lambda_SommerfeldBC_real_3D_z': 10.},
                               fixed_var=False,
                               batch_size_per_area=100)
        self.add(Wall, name="ABC")

        # interior
        interior = rec.interior_bc(outvar_sympy={'Maxwell_Freq_real_3D_x': 0.,
                                                 'Maxwell_Freq_real_3D_y': 0.,
                                                 'Maxwell_Freq_real_3D_z': 0.},
                                   bounds={x: (-width/2, width/2), y: (-length/2, length/2), z: (-height/2,
    height/2)},
                                   lambda_sympy={'lambda_Maxwell_Freq_real_3D_x': 1.0/wave_number**2,
                                                 'lambda_Maxwell_Freq_real_3D_y': 1.0/wave_number**2,
                                                 'lambda_Maxwell_Freq_real_3D_z': 1.0/wave_number**2},
                                   fixed_var=False,
                                   batch_size_per_area=1000)
        self.add(interior, name="Interior")
```

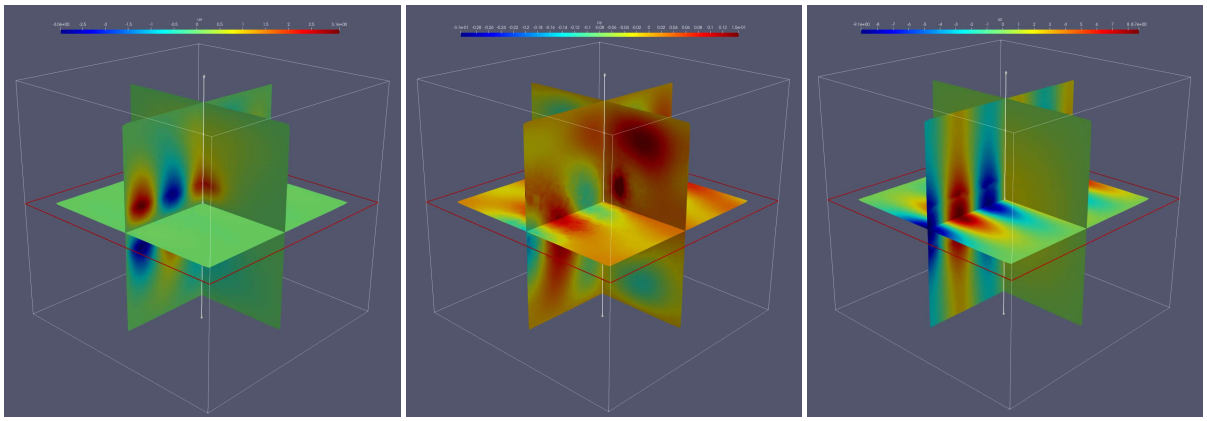
Listing 84: Define the TrainDomain

Finally, we define the InferenceDomain and Solver. These are same as the previous example except the bounds for the domain. The result is shown in Figure 51.

10.5.2 Results

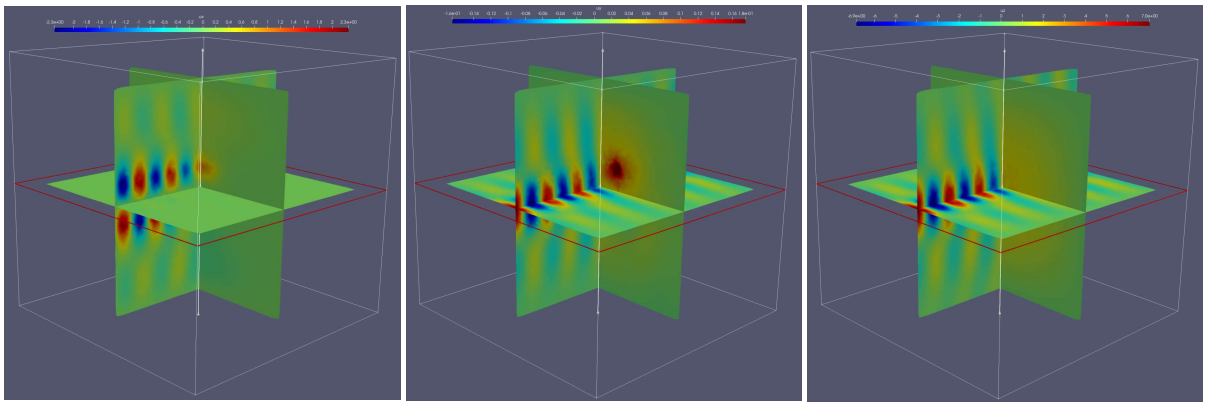
The full code of this example can be found in examples/waveguide/dielectric_slab_waveguide_3D.py. We do the simulation for different wavennumbers.

We also display the results of higher wavenumber 32 in Figure 52.



(a) 3D dielectric slab, E_x (b) 3D dielectric slab, E_y (c) 3D dielectric slab, E_z

Figure 51: Results for wavenumber 16



(a) 3D dielectric slab, E_x (b) 3D dielectric slab, E_y (c) 3D dielectric slab, E_z

Figure 52: Results for wavenumber 32

11 Linear Elasticity

11.1 Introduction

In this tutorial, we will walk you through the linear elasticity implementation in SimNet. SimNet offers the capability of solving the linear elasticity equations in the differential or variational form, allowing to solve a wide range of problems with a variety of boundary conditions. We present three examples in this chapter, namely the 3D bracket, the fuselage panel, and the plane displacement, to discuss the details of the linear elasticity in SimNet. Thus, in this tutorial, you would learn the following:

- How to solve linear elasticity equations using the differential and variational forms.
- How to solve linear elasticity problems for 3d and thin 2d structures (plane-stress).
- How to non-dimensionalize the elasticity equations.

11.2 Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity and have familiarized yourself with the basics of the SimNet user interface. Also, we recommend you to refer Section 1.7 from the Theory chapter for more details on weak solutions of PDEs. Some of the boundary conditions are defined more elaborately in the tutorial 9 and we encourage you to refer that tutorial for more details.

Note: The linear elasticity equations in Simnet can be found in the source code directory `simnet/PDES/linear_elasticity.py`.

11.3 Linear Elasticity in the Differential Form

11.3.1 Linear elasticity equations in the displacement form

The displacement form of the (non-transient) linear elasticity equations, known as the Navier equations, is defined as

$$(\lambda + \mu)u_{j,ji} + \mu u_{i,jj} + f_i = 0, \tag{87}$$

where u_i is the displacement vector, f_i is the body force per unit volume, and λ, μ are the Lamé parameters defined as

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \tag{88}$$

$$\mu = \frac{E}{2(1+\nu)}. \tag{89}$$

Here, E, v are, respectively, the Young's modulus and Poisson's ratio.

11.3.2 Linear elasticity equations in the mixed form

In addition to the displacement formulation, linear elasticity can also be described by the mixed-variable formulation. In fact, based on our experiments and also the studies reported in [33], the mixed-variable formulation is easier for a neural network solver to learn, possibly due to the lower order differential terms. In the mixed-variable formulation, the equilibrium equations are defend as

$$\sigma_{ji,j} + f_i = 0, \tag{90}$$

where σ_{ij} is the Cauchy stress tensor. the stress-displacement equations are also defined as

$$\sigma_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij}, \tag{91}$$

where δ_{ij} is the Kronecker delta function and ϵ_{ij} is the strain tensor that takes the following form

$$\epsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right). \tag{92}$$

11.3.3 Non-dimensionalized linear elasticity equations

It is often advantageous to work with the non-dimensionalized and normalized variation of the elasticity equations to improve the training convergence and accuracy. To this end, let us define our non-dimensionalized variables as

$$\hat{x}_i = \frac{x_i}{L}, \tag{93}$$

$$\hat{u}_i = \frac{u_i}{U}, \tag{94}$$

$$\hat{\lambda} = \frac{\lambda}{\mu_c}, \tag{95}$$

$$\hat{\mu} = \frac{\mu}{\mu_c} \tag{96}$$

Here, L is the characteristic length, U is the characteristic displacement, and μ_c is the non-dimensionalizing shear modulus. One can obtain the non-dimensionalized Navier and equilibrium equations by multiplying both sides of equations by $L^2/\mu_c U$, as follows

$$(\hat{\lambda} + \hat{\mu})\hat{u}_{j,ji} + \hat{\mu}\hat{u}_{i,jj} + \hat{f}_i = 0, \tag{97}$$

$$\hat{\sigma}_{ji,j} + \hat{f}_i = 0, \tag{98}$$

where the non-dimensionalized body force and stress tensor are

$$\hat{f}_i = \frac{L^2}{\mu_c U} f_i, \tag{99}$$

$$\hat{\sigma}_{ij} = \frac{L}{\mu_c U}\sigma_{ij}. \tag{100}$$

Similarly, the non-dimensionalized form of the stress-displacement equations are obtained by multiplying both sides of equations by $L/\mu_c U$, as follows

$$\hat{\sigma}_{ij} = \hat{\lambda}\hat{\epsilon}_{kk}\delta_{ij} + 2\hat{\mu}\hat{\epsilon}_{ij}, \tag{101}$$

$$\hat{\epsilon}_{ij} = \frac{1}{2}\left(\hat{u}_{i,j} + \hat{u}_{j,i}\right). \tag{102}$$

11.3.4 Plane stress equations

In a plane stress setting for thin structures, we assume that

$$\hat{\sigma}_{zz} = \hat{\sigma}_{xz} = \hat{\sigma}_{yz} = 0, \tag{103}$$

and therefore, the following relationship holds

$$\hat{\sigma}_{zz} = \hat{\lambda}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}} + \frac{\partial \hat{w}}{\partial \hat{z}}\right) + 2\hat{\mu}\frac{\partial \hat{w}}{\partial \hat{z}} = 0 \Rightarrow \frac{\partial \hat{w}}{\partial \hat{z}} = \frac{-\hat{\lambda}}{(\hat{\lambda} + 2\hat{\mu})}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}}\right). \tag{104}$$

Accordingly, the equations for $\hat{\sigma}_{xx}$ and $\hat{\sigma}_{yy}$ can be updated as follows

$$\hat{\sigma}_{xx} = \hat{\lambda}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}} + \frac{-\hat{\lambda}}{(\hat{\lambda} + 2\hat{\mu})}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}}\right)\right) + 2\hat{\mu}\frac{\partial \hat{u}}{\partial \hat{x}} \tag{105}$$

$$\hat{\sigma}_{yy} = \hat{\lambda}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}} + \frac{-\hat{\lambda}}{(\hat{\lambda} + 2\hat{\mu})}\left(\frac{\partial \hat{u}}{\partial \hat{x}} + \frac{\partial \hat{v}}{\partial \hat{y}}\right)\right) + 2\hat{\mu}\frac{\partial \hat{v}}{\partial \hat{y}}. \tag{106}$$

11.3.5 Problem 1: Deflection of a bracket

In this first linear elasticity example, we consider a 3D bracket as shown in Figure 53. This example is partially adopted form the MATLAB PDE toolbox. The back face of this bracket is clamped, and a traction of 4×10^4 Pa is applied to the front face in the negative-z direction (this face is shown in red). The rest of the surface is considered as traction-free boundaries. We set $(E, \nu) = (100\,\text{GPa}, 0.3)$. We non-dimensionalize the linear elasticity equations by setting $L = 1\,\text{m}, U = 0.0001\,\text{m}, \mu_c = 0.01\mu$:

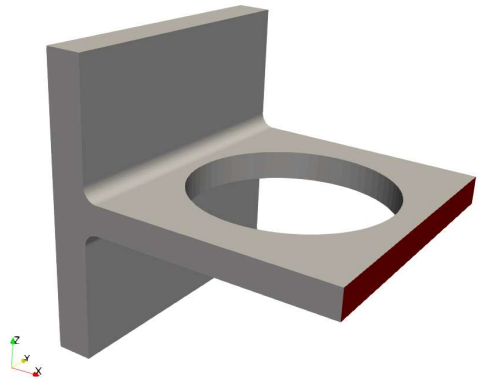


Figure 53: Geometry of the bracket. The back face of the bracket is clamped, and a shear stress is applied to the front face in the negatize-z direction.

```
# Parameters
nu                           = 0.3
E                            = 100e9
lambda_                      = nu*E/((1+nu)*(1-2*nu))
mu                           = E/(2*(1+nu))
mu_c                         = 0.01*mu
lambda_                      = lambda_/mu_c
mu                           = mu/mu_c
characteristic_length        = 1.
characteristic_displacement  = 1e-4
sigma_normalization          = characteristic_length/(characteristic_displacement*mu_c)
T                            = -4e4 * sigma_normalization
```

Listing 85: Non-dimensionalization for the bracket example.

As a rule of thumb, the characteristic length can be chosen in such a way to bound the largest dimension of the geometry to $(-1, 1)$. The characteristic displacement and μ_c can also be chosen such that the maximum displacement and the applied traction are close to 1 in order. Although the maximum displacement is not known a priori, we have observed that the convergence is not sensitive to the choice of the characteristic displacement as long as it is reasonably selected based on an initial guess for the approximate order of displacement. More information on non-dimensionalizing the PDEs can be found in Scaling of Differential Equations.

11.3.5.1 Case Setup and Results

We use the mixed-form of the linear elasticity equations here in this example, and therefore, we define the training domain as shown below. The complete python script for this problem can be found at `examples/bracket/bracket.py`.

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

    backBC = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0, 'w': 0},
                             lambda_sympy={'lambda_u': 10,
                                           'lambda_v': 10,
                                           'lambda_w': 10},
                             batch_size_per_area=2000,
                             criteria= Eq(x, support_origin[0]))
    self.add(backBC, name="backBC")

    frontBC = geo.boundary_bc(outvar_sympy={'traction_x': 0,
                                            'traction_y': 0,
                                            'traction_z': T},
```

```
                                    lambda_sympy={'lambda_traction_x': 1,
                                                  'lambda_traction_y': 1,
                                                  'lambda_traction_z': 1},
                                    batch_size_per_area=2000,
                                    criteria=Eq(x, bracket_origin[0]+bracket_dim[0]))
    self.add(frontBC, name="frontBC")

    surfaceBC = geo.boundary_bc(outvar_sympy={'traction_x': 0,
                                              'traction_y': 0,
                                              'traction_z': 0},
                                lambda_sympy={'lambda_traction_x': 1,
                                              'lambda_traction_y': 1,
                                              'lambda_traction_z': 1},
                                batch_size_per_area=2000,
                                criteria= (x>support_origin[0]) &
                                          (x<bracket_origin[0]+bracket_dim[0]))
    self.add(surfaceBC, name="surfaceBC")

    interior = geo.interior_bc(outvar_sympy={'equilibrium_x': 0.,
                                             'equilibrium_y': 0.,
                                             'equilibrium_z': 0.,
                                             'stress_disp_xx': 0.,
                                             'stress_disp_yy': 0.,
                                             'stress_disp_zz': 0.,
                                             'stress_disp_xy': 0.,
                                             'stress_disp_xz': 0.,
                                             'stress_disp_yz': 0.},
                               lambda_sympy={'lambda_equilibrium_x': geo.sdf,
                                             'lambda_equilibrium_y': geo.sdf,
                                             'lambda_equilibrium_z': geo.sdf,
                                             'lambda_stress_disp_xx': geo.sdf,
                                             'lambda_stress_disp_yy': geo.sdf,
                                             'lambda_stress_disp_zz': geo.sdf,
                                             'lambda_stress_disp_xy': geo.sdf,
                                             'lambda_stress_disp_xz': geo.sdf,
                                             'lambda_stress_disp_yz': geo.sdf},
                               bounds={x: bounds_support_x,
                                       y: bounds_support_y,
                                       z: bounds_support_z},
                               batch_size_per_area=16000)
    self.add(interior, name="Interior_support")

    interior = geo.interior_bc(outvar_sympy={'equilibrium_x': 0.,
                                             'equilibrium_y': 0.,
                                             'equilibrium_z': 0.,
                                             'stress_disp_xx': 0.,
                                             'stress_disp_yy': 0.,
                                             'stress_disp_zz': 0.,
                                             'stress_disp_xy': 0.,
                                             'stress_disp_xz': 0.,
                                             'stress_disp_yz': 0.},
                               lambda_sympy={'lambda_equilibrium_x': geo.sdf,
                                             'lambda_equilibrium_y': geo.sdf,
                                             'lambda_equilibrium_z': geo.sdf,
                                             'lambda_stress_disp_xx': geo.sdf,
                                             'lambda_stress_disp_yy': geo.sdf,
                                             'lambda_stress_disp_zz': geo.sdf,
                                             'lambda_stress_disp_xy': geo.sdf,
                                             'lambda_stress_disp_xz': geo.sdf,
                                             'lambda_stress_disp_yz': geo.sdf},
                               bounds={x: bounds_bracket_x,
                                       y: bounds_bracket_y,
                                       z: bounds_bracket_z},
                               batch_size_per_area=16000)
    self.add(interior, name="Interior_bracket")
```

Listing 86: Training domain for the bracket example.

Note that the training domain consists of two different sets of interior points (i.e., `Interior_support` and `Interior_bracket`). This is done only to generate the interior points more efficiently. We use two separate neural networks for displacement and stresses, as follows

```
def __init__(self, **config):
  super(BracketSolver, self).__init__(**config)
  self.equations = (LinearElasticity(lambda_=lambda_, mu=mu, dim=3).make_node())
  elasticity_net_disp = self.arch.make_node(name='elasticity_net_disp',
                                            inputs=['x', 'y', 'z'],
                                            outputs=['u', 'v', 'w'])
  elasticity_net_stress = self.arch.make_node(name='elasticity_net_stress',
```

```
                                            inputs=['x', 'y', 'z'],
                                            outputs=['sigma_xx', 'sigma_yy',
                                                     'sigma_zz', 'sigma_xy',
                                                     'sigma_xz', 'sigma_yz'])
  self.nets = [elasticity_net_disp, elasticity_net_stress]
```

Listing 87: Defining the PDEs and networks for the bracket example.

Figure 56 shows the SimNet results and also a comparison with a commercial solver results. The results of these two solvers show good agreement, with only a 8% difference in maximum bracket displacement.

11.3.6 Problem 2: Stress analysis for aircraft fuselage panel

In this example, we will use SimNet for the analysis of stress concentration in an aircraft fuselage panel. Depending on the altitude of the flying plane, the fuselage panel is exposed to different values of hoop stress that can cause accumulated damage to the panel over the time. Therefore, in cumulative damage modeling of aircraft fuselage for the purpose of design and maintenance of aircrafts, it is required to perform several stress simulations for different hoop stress values. Here, we consider a simplified aircraft fuselage panel as shown in Figure 55, and construct a parameterized model that, once trained, can predict the stress and displacement in the panel for different values of hoop stress.

11.3.6.1 Case Setup and Results The panel material is Aluminium 2024-T3, with $(E, \nu) = (73\,\text{GPa}, 0.33)$. We train a parameterized model with varying $\sigma_{hoop} \in (46, 56.5)$. Since the thickness of the panel is very small (i.e., 2 mm), we will solve the plane stress equations in this example. The python script for this problem can be found at `examples/fuselage_panel/panel_solver.py`. The plane stress form of the linear elasticity equations in SimNet can be called by using the `LinearElasticityPlaneStress` class:

```
self.equations = (LinearElasticityPlaneStress(lambda_=lambda_, mu=mu).make_node())
```

Listing 88: Using the plane stress equations in SimNet.

Figure 56 shows the SimNet results for panel displacements and stresses. For comparison, the commercial solver results are also presented in Figure 57. The SimNet and commercial solver results are in close agreement, with a difference of less than 5% in the maximum Von Mises stress.

11.4 Linear Elasticity in the Variational Form

11.4.1 Linear elasticity equations in the variational form

In addition to the differential form of the linear elasticity equations, SimNet also enables the use of variational form of these equations (Tutorial 9). In this section, we derive the linear elasticity equations in the variational form as implemented in SimNet. We will use the non-dimensionalized variables in this derivation. We start by forming the inner product of Equation 98 and a vector test function $v \in \mathbf{V}$, and integrating over the domain, as follows

$$\int_{\Omega} \hat{\sigma}_{ji,j} v_i d\mathbf{x} + \int_{\Omega} \hat{f}_i v_i d\mathbf{x} = 0. \tag{107}$$

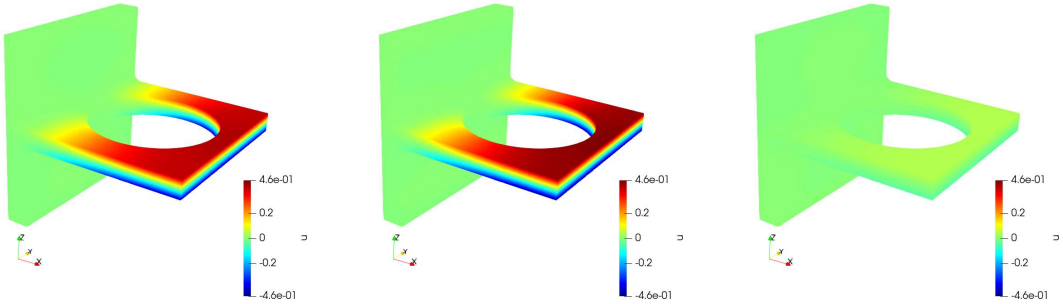
Using the integration by parts, we have

$$\int_{\partial\Omega} \hat{T}_i v_i d\mathbf{s} - \int_{\Omega} \hat{\sigma}_{ji} v_{j,i} d\mathbf{x} + \int_{\Omega} \hat{f}_i v_i d\mathbf{x} = 0, \tag{108}$$

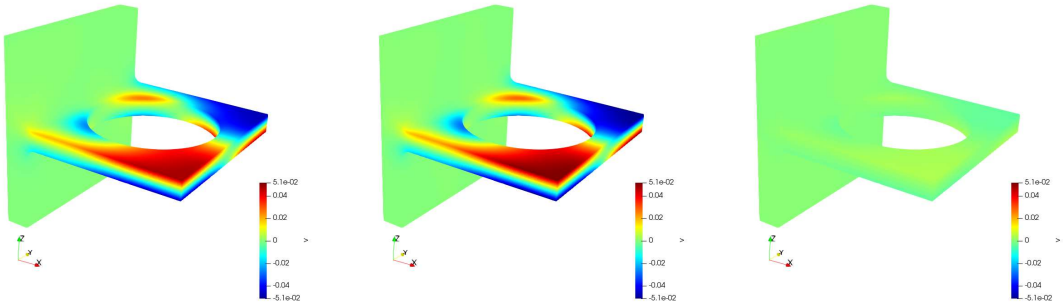
where T_i is the traction. Note that the first term is zero for the traction-free boundaries. Equation 108 is the variational form of the linear elasticity equations that is adopted in SimNet. The term $\hat{\sigma}_{ji}$ in this equation is computed using the stress-displacement relation in Equation 101.

11.4.2 Problem 3: Plane displacement

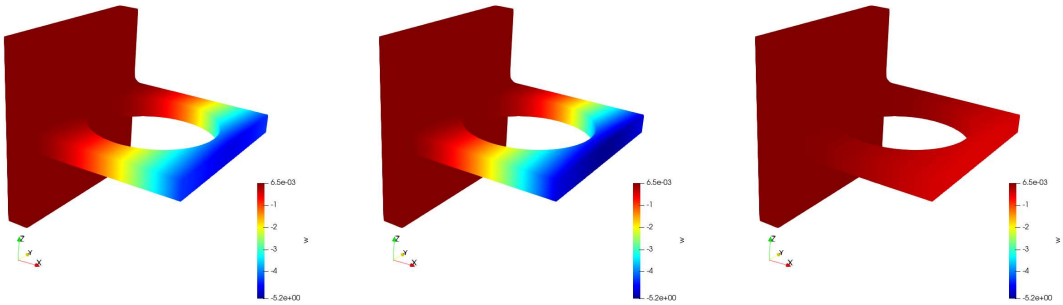
In this example, we solve the linear elasticity plane stress equations in the variational form. We consider a square plate that is clamped from one end and is under a displacement boundary condition on the other end, as shown in Figure 58. The rest of the boundaries are traction-free. The material properties are assumed to be $(E, \nu) = (10\,\text{MPa}, 0.2)$. This example is adopted from [33].



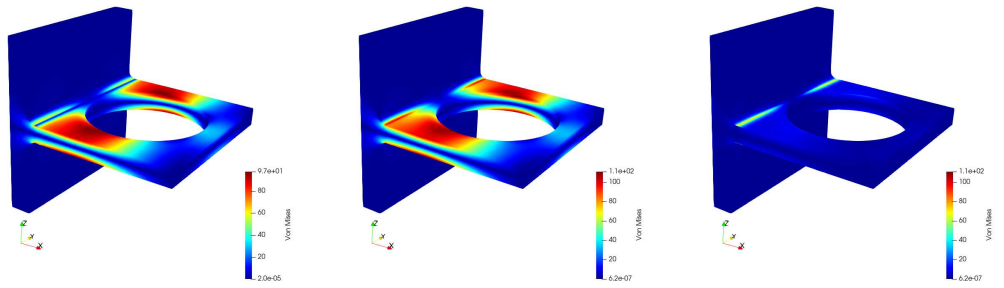
(a) u (SimNet) (b) u (Commercial solver) (c) u (Difference)



(d) v (SimNet) (e) v (Commercial solver) (f) v (Difference)



(g) w (SimNet) (h) w (Commercial Solver) (i) w (Difference)



(j) σ_{VM} (SimNet) (k) σ_{VM} (Commercial Solver) (l) σ_{VM} (Difference)

Figure 54: SimNet linear elasticity results for the bracket deflection example.

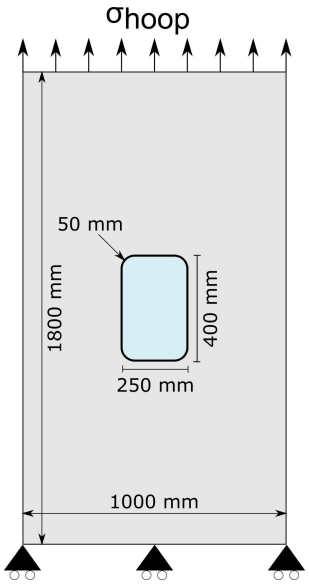
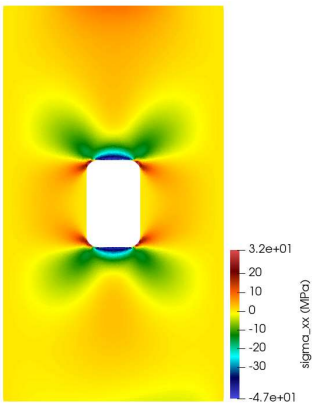
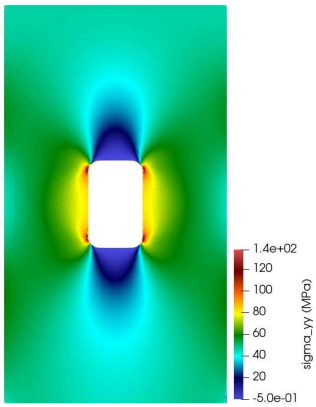


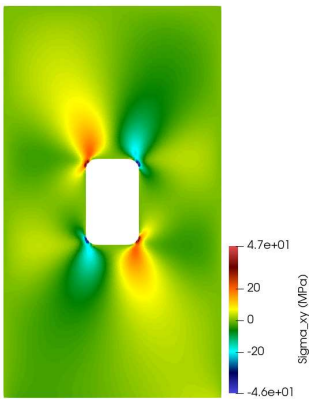
Figure 55: Geometry and boundary conditions of the simplified aircraft fuselage panel.



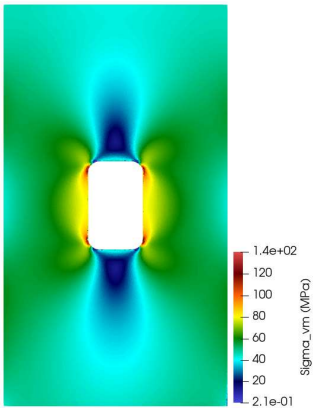
(a) σ_{xx}



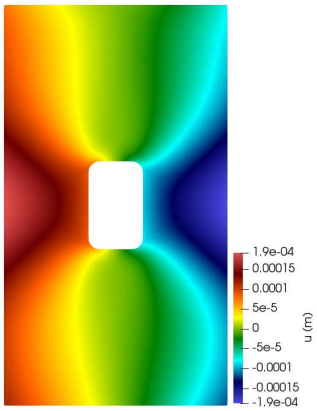
(b) σ_{yy}



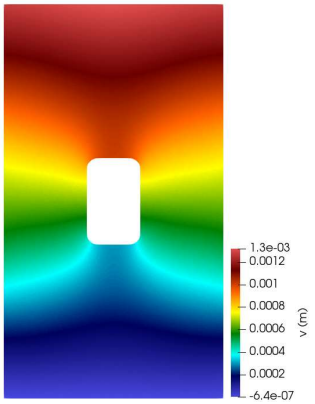
(c) σ_{xy}



(d) Von Mises stress

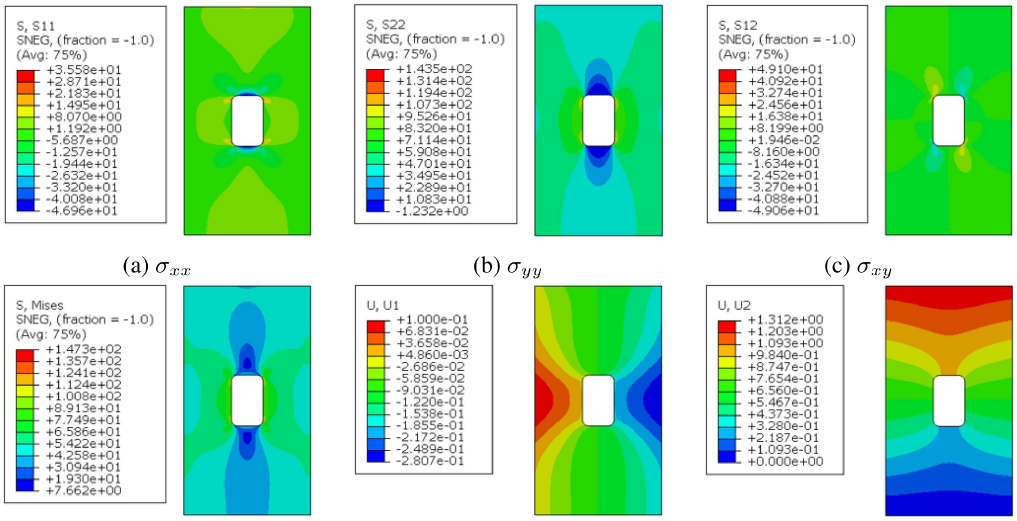


(e) u



(f) v

Figure 56: SimNet linear elasticity results for the aircraft fuselage panel example with parameterized hoop stress. The results are for $\sigma_{hoop} = 46$.



(a) σ_{xx} (b) σ_{yy} (c) σ_{xy}

(d) Von Mises stress (e) u (f) v

Figure 57: commercial solver linear elasticity results for the aircraft fuselage panel example with $\sigma_{hoop} = 46$.

11.4.2.1 Case Setup and Results To solve this problem in the variational form, we enforce the displacement boundary conditions in the differential form, and also generate interior and boundary points to be used in evaluation of the integrals in Equation 108. The python script for this problem can be found at `examples/plane_displacement/plane_displacement.py`.

```
class PlaneDisplacementTrain(TrainDomain):
  def __init__(self, **config):
    super(PlaneDisplacementTrain, self).__init__()
    bottomBC = geo.boundary_bc(outvar_sympy={'u': 0., 'v': 0.},
                               lambda_sympy={'lambda_u': 10,
                                             'lambda_v': 10},
                               batch_size_per_area=500,
                               criteria=Eq(y, domain_origin[1]),
                               batch_per_epoch=5000)
    self.add(bottomBC, name="bottomBC_differential")

    bottomBC = geo.boundary_bc(outvar_sympy={},
                               batch_size_per_area=500,
                               criteria=Eq(y, domain_origin[1]),
                               fixed_var=False,
                               quasirandom=True)
    self.add(bottomBC, name="bottomBC")

    topBC = geo.boundary_bc(outvar_sympy={'u': 0., 'v': 0.1},
                            lambda_sympy={'lambda_u': 10,
                                          'lambda_v': 10},
                            batch_size_per_area=500,
                            criteria=Eq(y, domain_origin[1]+ domain_dim[1])
                                     & (x<= domain_origin[0]+ domain_dim[0]/2.),
                            batch_per_epoch=5000)
    self.add(topBC, name="topBC_differential")

    topBC = geo.boundary_bc(outvar_sympy={},
                            batch_size_per_area=500,
                            criteria=Eq(y, domain_origin[1]+ domain_dim[1])
                                     & (x<= domain_origin[0]+ domain_dim[0]/2.),
                            fixed_var=False,
                            quasirandom=True)
    self.add(topBC, name="topBC")

    interior = geo.interior_bc(outvar_sympy={},
                               batch_size_per_area=10000,
                               bounds={x: bounds_x, y: bounds_y},
                               fixed_var=False,
                               quasirandom=True)
    self.add(interior, name="Interior")
```

Listing 89: Training domain for the plane displacement example.

Note that for the training points that are used in the variational form, we set the `outvar_sympy` to be an empty dictionary.

So far, we have only included the displacement boundary conditions in our loss function. We will define a custom loss function that includes the variational loss, and add that to the overall loss. In the remainder of this subsection, we will discuss how to generate this variational loss. We will first compute the neural network solution at the interior and boundary points, and then compute the required gradients:

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

  # compute solution for the interior
  u_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['u']
  u_x_interior = tf.gradients(u_interior, x_interior)[0]
  u_y_interior = tf.gradients(u_interior, y_interior)[0]
  v_interior = self.nets[0].evaluate({'x': x_interior, 'y': y_interior})['v']
  v_x_interior = tf.gradients(v_interior, x_interior)[0]
  v_y_interior = tf.gradients(v_interior, y_interior)[0]

  # get points on the boundary
  x_bottom_dir = domain_invar['bottomBC']['x']
  y_bottom_dir = domain_invar['bottomBC']['y']
  normal_x_bottom_dir = domain_invar['bottomBC']['normal_x']
  normal_y_bottom_dir = domain_invar['bottomBC']['normal_y']
  area_bottom_dir = domain_invar['bottomBC']['area']

  x_top_dir = domain_invar['topBC']['x']
  y_top_dir = domain_invar['topBC']['y']
  normal_x_top_dir = domain_invar['topBC']['normal_x']
  normal_y_top_dir = domain_invar['topBC']['normal_y']
  area_top_dir = domain_invar['topBC']['area']

  # compute solution for the boundary
  u_bottom_dir = self.nets[0].evaluate({'x': x_bottom_dir, 'y': y_bottom_dir})['u']
  u_x_bottom_dir = tf.gradients(u_bottom_dir, x_bottom_dir)[0]
  u_y_bottom_dir = tf.gradients(u_bottom_dir, y_bottom_dir)[0]
  v_bottom_dir = self.nets[0].evaluate({'x': x_bottom_dir, 'y': y_bottom_dir})['v']
  v_x_bottom_dir = tf.gradients(v_bottom_dir, x_bottom_dir)[0]
  v_y_bottom_dir = tf.gradients(v_bottom_dir, y_bottom_dir)[0]

  u_top_dir = self.nets[0].evaluate({'x': x_top_dir, 'y': y_top_dir})['u']
  u_x_top_dir = tf.gradients(u_top_dir, x_top_dir)[0]
  u_y_top_dir = tf.gradients(u_top_dir, y_top_dir)[0]
  v_top_dir = self.nets[0].evaluate({'x': x_top_dir, 'y': y_top_dir})['v']
  v_x_top_dir = tf.gradients(v_top_dir, x_top_dir)[0]
  v_y_top_dir = tf.gradients(v_top_dir, y_top_dir)[0]
```

Listing 90: Computing the neural network solution and its gradients at the interior and boundary points for the plane displacement example.

In the next step, we define our test functions, and compute the test functions and their required gradients at the interior and boundary points:

```
  # test functions
  test_fn = Test_Function(name_ord_dict={Legendre_test: [k for k in range(10)],
                                         Trig_test: [k for k in range(10)]},
                                         box =[[domain_origin[0], domain_origin[1]],
                                               [domain_origin[0]+domain_dim[0],
                                                domain_origin[1]+domain_dim[1]]],
                                         diff_list=['grad'])

  v = Vector_Test(test_fn,test_fn,mix=0.02)
  vx_x_interior, vy_x_interior = v.eval_test('vx', x_interior, y_interior)
  vx_y_interior, vy_y_interior = v.eval_test('vy', x_interior, y_interior)
  vx_bottom_dir, vy_bottom_dir = v.eval_test('v', x_bottom_dir, y_bottom_dir)
  vx_top_dir, vy_top_dir = v.eval_test('v', x_top_dir, y_top_dir)
```

Listing 91: Defining the test functions for the plane displacement example.

Here, we construct a set of test functions consisting of Legendre polynomials and trigonometric functions, and randomly subsample 2% of these test functions. Note that we only compute the terms that appear in the variational loss in

Equation 108. For instance, it is not necessary to compute the derivative of the test functions with respect to input coordinates for the boundary points.

The next step is to compute the stress terms according to the plane stress equations in Equations 105, 106, and also the traction terms:

```
    w_z_interior          = -lambda_/(lambda_+2*mu) * (u_x_interior+v_y_interior)
    sigma_xx_interior     = lambda_ * (u_x_interior+v_y_interior+w_z_interior)
                            + 2*mu * u_x_interior
    sigma_yy_interior     = lambda_ * (u_x_interior+v_y_interior+w_z_interior)
                            + 2*mu * v_y_interior
    sigma_xy_interior     = mu * (u_y_interior+v_x_interior)

    w_z_bottom_dir        = -lambda_/(lambda_+2*mu) * (u_x_bottom_dir+v_y_bottom_dir)
    sigma_xx_bottom_dir   = lambda_ * (u_x_bottom_dir+v_y_bottom_dir+w_z_bottom_dir)
                            + 2*mu * u_x_bottom_dir
    sigma_yy_bottom_dir   = lambda_ * (u_x_bottom_dir+v_y_bottom_dir+w_z_bottom_dir)
                            + 2*mu * v_y_bottom_dir
    sigma_xy_bottom_dir   = mu * (u_y_bottom_dir+v_x_bottom_dir)

    w_z_top_dir           = -lambda_/(lambda_+2*mu) * (u_x_top_dir+v_y_top_dir)
    sigma_xx_top_dir      = lambda_ * (u_x_top_dir+v_y_top_dir+w_z_top_dir)
                            + 2*mu * u_x_top_dir
    sigma_yy_top_dir      = lambda_ * (u_x_top_dir+v_y_top_dir+w_z_top_dir)
                            + 2*mu * v_y_top_dir
    sigma_xy_top_dir      = mu * (u_y_top_dir+v_x_top_dir)

    traction_x_bottom_dir = sigma_xx_bottom_dir*normal_x_bottom_dir
                            +sigma_xy_bottom_dir*normal_y_bottom_dir
    traction_y_bottom_dir = sigma_xy_bottom_dir*normal_x_bottom_dir
                            +sigma_yy_bottom_dir*normal_y_bottom_dir
    traction_x_top_dir    = sigma_xx_top_dir*normal_x_top_dir
                            +sigma_xy_top_dir*normal_y_top_dir
    traction_y_top_dir    = sigma_xy_top_dir*normal_x_top_dir
                            +sigma_yy_top_dir*normal_y_top_dir
```

Listing 92: Computing the stress and traction terms for the plane displacement example.

Finally, following the Equation 108, we define our variational interior and boundary integral terms, formulate the variational loss, and add that to the overall loss, as follows

```
    interior_loss = tensor_int(area_interior, sigma_xx_interior*vx_x_interior
                                            +sigma_yy_interior*vy_y_interior
                                            +sigma_xy_interior*(vx_y_interior
                                                               +vy_x_interior))
    boundary_loss1 = tensor_int(area_bottom_dir, traction_x_bottom_dir*vx_bottom_dir
                                               +traction_y_bottom_dir*vy_bottom_dir)
    boundary_loss2 = tensor_int(area_top_dir, traction_x_top_dir*vx_top_dir
                                           +traction_y_top_dir*vy_top_dir)

    # make variational loss function
    loss = Variables()
    loss['loss_variational'] = tf.reduce_sum(tf.square(interior_loss - boundary_loss1 - boundary_loss2))
    tf.summary.scalar('loss_variational', loss['loss_variational'])
    return loss
```

Listing 93: Constructing the variational loss for the plance displacement example.

Figure 58 shows the SimNet results for this plane displacement example. The results are in good agreement with the FEM results reported in [33], verifying the accuracy of the SimNet results in the variational form.

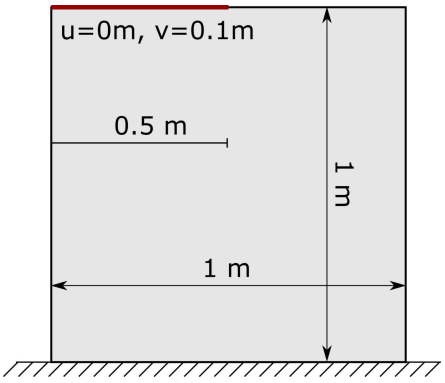
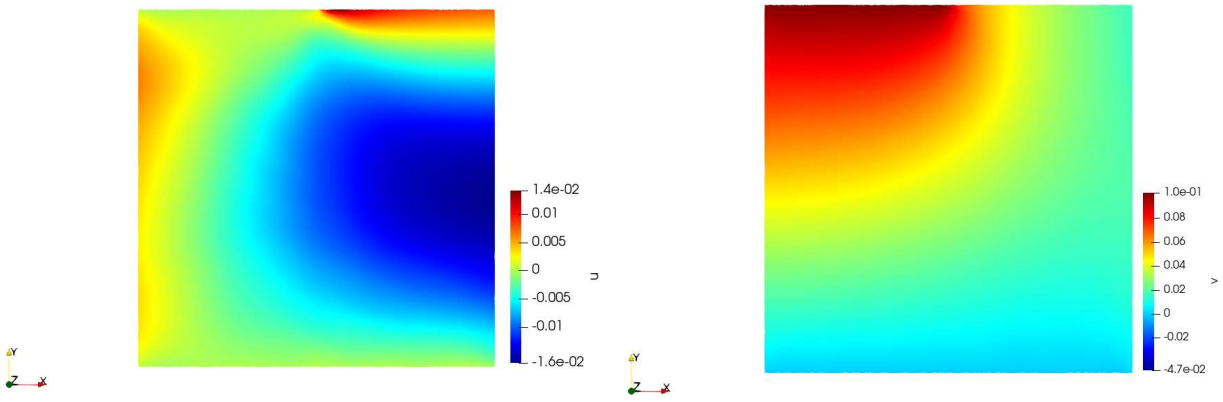


Figure 58: Geometry and boundary conditions of the plane displacement example. This example is adopted from [33].



(a) u

(b) v

Figure 59: SimNet results for the plane displacement example.

12 Tuning Neural Network Hyperparameters & Using SimNet's Advanced Features

12.1 Introduction

In this tutorial, we will briefly cover the sections of SimNet library which you can refer to make any modifications to the Neural Network architecture, implement some of the advanced SimNet features as well as perform hyper-parameter tuning.

Note: Unlike other tutorials, we will not cover the details of the code implementation. This tutorial will serve as a one-stop guide for all the important parameters that are at your disposal for hyperparameter tuning from an application perspective. However, for any details on the theory or the code implementation, we encourage you to visit the tutorial 1 and the source code documentation respectively.

Let's get started.

Most of the changes like modifying the number of hidden layers, activation function, learning rate parameters, etc. can be modified by `update_defaults` class method while creating the `Solver` class (Base class can be found at `simnet/solver.py`) or by passing flags while executing the command. We have already used the `update_defaults` method for specifying the network directory to write, max steps for the problem, etc. in previous tutorials.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_hemholtz',
        'rec_results': True,
        'rec_results_freq': 1000,
        'start_lr': 3e-4,
        'decay_steps': 20000,
        'max_steps': 40000
        })
```

Listing 94: Update defaults

The same can be achieved by the following command while executing the script.

```
python helmholtz.py --rec_results=True --rec_results_freq=1000 --start_lr=3e-4 --
    decay_steps=20000
```

To see the available options for hyper-parameter tuning, one can refer to the source code documentation or simply run the python script with `-h` flag.

```
python filename.py -h
```

The list of options should be displayed in the command window. An example is shown below:

```
usage: ldc_2d.py [-h] [--run_mode {solve,eval,plot_data}]
                 [--network_dir NETWORK_DIR]
                 [--initialize_network_dir INITIALIZE_NETWORK_DIR]
                 [--added_config_dir ADDED_CONFIG_DIR]
                 [--rec_results REC_RESULTS]
                 [--rec_results_freq REC_RESULTS_FREQ] [--max_steps MAX_STEPS]
                 [--save_filetypes SAVE_FILETYPES] [--xla XLA]
                 [--inner_norm INNER_NORM] [--outer_norm OUTER_NORM]
                 [--activation_fn ACTIVATION_FN] [--layer_size LAYER_SIZE]
                 [--nr_layers NR_LAYERS] [--skip_connections SKIP_CONNECTIONS]
                 [--weight_norm WEIGHT_NORM] [--start_lr START_LR]
                 [--end_lr END_LR] [--decay_steps DECAY_STEPS]
                 [--decay_rate DECAY_RATE] [--beta1 BETA1] [--beta2 BETA2]
                 [--epsilon EPSILON] [--amp AMP]

optional arguments:
  -h, --help            show this help message and exit

Controller Details:
  --run_mode {solve,eval,tkinter,plot_data}
                        all modes

Solver Configs:
  --network_dir NETWORK_DIR
                        where to save neural network solver
  --initialize_network_dir INITIALIZE_NETWORK_DIR
                        restore weights from this dir. This can be used to
                        solve one-way coupled systems sequentially for example
  --added_config_dir ADDED_CONFIG_DIR
```

```
                        configs to add onto network_dir seperated by comma
  --rec_results REC_RESULTS
                        record results while training
  --rec_results_freq REC_RESULTS_FREQ
                        steps till recording results
  --max_steps MAX_STEPS
                        max train steps
  --save_filetypes SAVE_FILETYPES
                        file types to save results too [vtk, csv, np]
  --xla XLA             use the XLA graph compiler?
  --inner_norm INNER_NORM
                        what norm to use to compute the loss for a single
                        field? (L1/L2)
  --outer_norm OUTER_NORM
                        what norm to use to compute the loss across fields?
                        (L1/L2)

Arch Configs:
  --activation_fn ACTIVATION_FN
                        nonlinearity for network
  --layer_size LAYER_SIZE
                        hidden layer size
  --nr_layers NR_LAYERS
                        nr layers in net
  --skip_connections SKIP_CONNECTIONS
                        residual skip connections
  --weight_norm WEIGHT_NORM
                        residual skip connections

LR Configs:
  --start_lr START_LR   start at this value
  --end_lr END_LR       decay to this value
  --decay_steps DECAY_STEPS
                        how many steps to take for decay
  --decay_rate DECAY_RATE
                        decay rate

Optimizer Configs:
  --beta1 BETA1         beta 1
  --beta2 BETA2         beta 2
  --epsilon EPSILON     epsilon
  --amp AMP             use Automatic Mixed Precision (AMP)? (0/1)
```

12.2 Network Architecture

The different architectures implemented in SimNet can be found in `simnet.architecture` module. Currently, SimNet has eight architectures for you to choose from: (1) fully-connected neural network (`fully_connected.py`), (2) Fourier network (`fourier_net.py`), (3) SiReN (`siren.py`), (4) Combination of Perdikaris architecture [10] and Fourier Networks (`modified_fourier_net.py`), (5) Combination of highway networks [11] and Fourier networks (`highway_fourier_net.py`), (6) DGM architecture (`dgm.py`), (7) multiplicative filter networks (`multiplicative_filter_net.py`), and (8) radial basis network (`radial_basis.py`).

A network is imported in from the `architecture` module and can be specified while instantiating the `Solver` class. An example is shown below.

```
from simnet.architecture.radial_basis import RadialBasisArch

# Define neural network
class ChipSolver(Solver):
  train_domain     = LDCTrain
  val_domain       = LDCVal
  arch             = RadialBasisArch
```

Listing 95: Selecting the architecture

The parameters to tune for the `FullyConnectedArch` are, `activation_fn`, `layer_size`, `nr_layers`, `skip_connections` and `weight_norm`. They can be modified in `update_defaults` as shown below.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'layer_size': 512,
        'nr_layer': 8,
        'skip_connections': False
        })
```

Listing 96: Changing the layers and layer size

For more information on the default values for these parameters, please refer to individual architecture files or the source code documentation.

12.3 Activation Functions

The complete list of activation functions available in SimNet can be found in the source code location `simnet/tf_utils/activation_functions.py`. The default activation function in SimNet is the swish but we also have implementations of activation functions like sin, tanh, etc.

The activation function can be changed as shown below.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'activation_fn': 'tanh'
        })
```

Listing 97: Changing activation functions

By default, SimNet does not use adaptive activation functions (described in Section 1.6.2.4). However, it is easy to turn on this feature, as shown below.

```
  def update_defaults(cls, defaults):
    defaults.update({
        'adaptive_activations': True
        })
```

Listing 98: Using adaptive activations

12.4 Learning Rate Schedule

The different learning rate schedules available in SimNet can be found in the source code location `simnet/learning_rate.py`. The default is an exponential decay, but you can choose among exponential decay, polynomial decay, exponential decay with warm-up or cyclic schedules for the learning rates.

An example to use polynomial decay is shown below.

```
from simnet.learning_rate import LR, PolynomialDecayLR

# Define neural network
class ChipSolver(Solver):
  train_domain     = LDCTrain
  val_domain       = LDCVal
  lr               = PolynomialDecayLR

  def update_defaults(cls, defaults):
    defaults.update({
        'start_lr': 3e-04,
        'decay_steps': 2000,
        'power': 0.5
        })
```

Listing 99: Changing learning rate schedule

For more information on the default values for these parameters, please refer to the source code documentation.

12.5 Optimizers

Currently, SimNet only has the Adam optimizer implemented. You can modify the optimizer settings or add your own implementation of optimizer in the source code at `simnet/optimizer.py` and source it appropriately.

For more information on the default values for the optimizer parameters, please refer to the source code documentation.

By default, SimNet does not use the global or local learning rate annealing (as described in Section 1.6.2.1). A sample for using the global learning rate annealing for the lid-driven cavity example (as described in Section 2) is shown below.

```
# import SimNet library
from simnet.optimizer import AdamOptimizerAnnealing

class LDCSolver(Solver):
  optimizer = AdamOptimizerAnnealing

  def __init__(self, **config):
    super(LDCSolver, self).__init__(**config)
    self.optimizer.grouping = [{'continuity'},
                               {'momentum_x','momentum_y'},
                               {'u','v','p'}]
  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'local_annealing': False
    })
```

Listing 100: Using the global learning rate annealing for the lid-driven cavity example.

In order to use the local learning rate annealing instead, you can simply set the `'local_annealing'` to `True`. In `self.optimizer.grouping`, we specify which loss terms will be grouped together. Please note that all of the defined loss terms should be included here.

12.6 Gradient Aggregation

By default, SimNet does not use gradient aggregation. The following shows how to activate gradient aggregation for the annular ring example by using the `AdamOptimizerGradAgg` class.

```
from simnet.optimizer import AdamOptimizerGradAgg
grad_agg_freq = 2 # number of gradient aggregations

class AnnularRingTrain(TrainDomain):
  def __init__(self, **config):
    super(AnnularRingTrain, self).__init__()
    # inlet
    inlet_sympy = parabola(y, inter_1=channel_width[0], inter_2=channel_width[1], height=inlet_vel)
    inlet = geo.boundary_bc(outvar_sympy={'u': inlet_sympy, 'v': 0},
                            batch_size_per_area=32,
                            batch_per_epoch = 1000*grad_agg_freq,
                            criteria=Eq(x, channel_length[0]))
    self.add(inlet, name="Inlet")

    # outlet
    outlet = geo.boundary_bc(outvar_sympy={'p': 0},
                             batch_size_per_area=32,
                             batch_per_epoch = 1000*grad_agg_freq,
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="Outlet")

    # noslip
    noslip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                             batch_size_per_area=32,
                             batch_per_epoch = 1000*grad_agg_freq,
                             criteria=(x>channel_length[0])&(x<channel_length[1]))
    self.add(noslip, name="NoSlip")

    # interior
    interior = geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0},
                               bounds={x: channel_length,
                                       y: (-outer_cylinder_radius, outer_cylinder_radius)},
                               lambda_sympy={'lambda_continuity': geo.sdf,
                                             'lambda_momentum_x': geo.sdf,
                                             'lambda_momentum_y': geo.sdf},
                               batch_size_per_area=128,
                               batch_per_epoch = 1000*grad_agg_freq)
    self.add(interior, name="Interior")

    # make integral continuity
    outlet = geo.boundary_bc(outvar_sympy={'integral_continuity': 2},
                             batch_size_per_area=128,
                             batch_per_epoch = 1000*grad_agg_freq
                             lambda_sympy={'lambda_integral_continuity': 0.1},
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="OutletContinuity")

class ChipSolver(Solver):
```

```
  train_domain = AnnularRingTrain
  val_domain = AnnularRingVal
  inference_domain = AnnularRingInference
  monitor_domain = AnnularRingMonitor
  optimizer = AdamOptimizerGradAgg

  def __init__(self, **config):
    super(ChipSolver, self).__init__(**config)
    self.equations = (NavierStokes(nu=0.01, rho=1, dim=2, time=False).make_node()
                      + IntegralContinuity(dim=2).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y'],
                                   outputs=['u', 'v', 'p'])
    self.nets = [flow_net]
    self.save_network_freq = 1000*grad_agg_freq
    self.print_stats_freq = 100*grad_agg_freq
    self.tf_summary_freq = 500*grad_agg_freq

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_annular_ring_grad_agg',
        'decay_steps': 2000*grad_agg_freq,
        'max_steps': 200000*grad_agg_freq,
        'grad_agg_freq': grad_agg_freq
        })
```

Listing 101: Using gradient aggregation.

Here, `grad_agg_freq` represents the number of gradient aggregations per model parameter update. Effectively, setting `grad_agg_freq=2` will double the batch size per iteration, compared to a case with no gradient aggregation. We also set the `batch_per_epoch` to `1000*grad_agg_freq` to scale the size of the fixed point cloud by the number of aggregations (the default value for `batch_per_epoch` is 1000). This will keep the ratio between the size of the fixed point cloud and the batch size the same as for the case with no gradient aggregation. Please note that when gradient aggregation is used, the training iteration counter is updated at every gradient aggregation step (and not only at each model parameter update). Therefore, to keep everything consistent with the case where no gradient aggregation is used, we will also scale `self.save_network_freq, self.print_stats_freq, self.tf_summary_freq, decay_steps, max_steps` by the number of gradient aggregations.

12.7 Importance Sampling

Here we demonstrate the importance sampling on the interior domain of the annular ring example. The script is available at `annular_ring/annular_ring_importance_sampling.py`. After a warmup phase that the training points are sampled uniformly, at every 100 iterations, we compute the velocity derivatives on a 1024×512 grid, and approximate the sampling probability proportional to the 2-norm of the velocity gradients on the interior using a nearest neighbor interpolation. The interior points are then sampled according to these sampling probabilities. As shown in Equation 19, a correction factor is multiplied by the loss at each training point to avoid bias in the loss estimation.

```
class AnnularRingTrain(TrainDomain):
  def __init__(self, **config):
    super(AnnularRingTrain, self).__init__(nr_threads=1)

    # inlet
    inlet_sympy = parabola(y, inter_1=channel_width[0], inter_2=channel_width[1], height=inlet_vel)
    inlet = geo.boundary_bc(outvar_sympy={'u': inlet_sympy, 'v': 0},
                            batch_size_per_area=32,
                            criteria=Eq(x, channel_length[0]))
    self.add(inlet, name="Inlet")

    # outlet
    outlet = geo.boundary_bc(outvar_sympy={'p': 0},
                             batch_size_per_area=32,
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="Outlet")

    # noslip
    noslip = geo.boundary_bc(outvar_sympy={'u': 0, 'v': 0},
                             batch_size_per_area=32,
                             criteria=(x>channel_length[0])&(x<channel_length[1]))
    self.add(noslip, name="NoSlip")

    # importance sampled interior
```

```
    # evaluate flow on grid for generating importance sampling probability function
    grid_res = [1024, 512]

    # make numpy function for computing the sdf
    sdf_fn = np_lambdify(geo.sdf,
                         ['x', 'y'])

    # make function for computing the network velocity derivatives
    # train domain now has access to sess, nets, and equations
    invar_placeholder = Variables.tf_variables(['x', 'y'],
                                               batch_size=grid_res[0]*grid_res[1])
    outvar_velocity_grad = unroll_graph(self.nets+self.equations,
                                        invar_placeholder,
                                        Key.convert_list(['u__x', 'u__y', 'v__x', 'v__y']))
    velocity_grad_fn = Variables.lambdify_np(invar_placeholder,
                                             outvar_velocity_grad, self.sess)

    # construct importance sampled constrain for interior
    class ImportanceSampledInterior(BC):
      def __init__(self):
        self.batch_size = 4000
        self.importance_prob_grid = None
        self.update_importance_prob_freq = 100
        self.sampled_counter = 0
        self.importance_prob_warmup = 25000
        self.epsilon_uniform = 0.

      def generate_importance_prob_grid(self):
        # sample 10 times as many points in batch to do adaptive sampling
        mesh_x, mesh_y = np.meshgrid(np.linspace(channel_length[0], channel_length[1], grid_res[0]),
                                     np.linspace(-outer_cylinder_radius, outer_cylinder_radius, grid_res[1]),
                                     indexing='ij')
        mesh_x = np.reshape(mesh_x, (grid_res[0]*grid_res[1], 1))
        mesh_y = np.reshape(mesh_y, (grid_res[0]*grid_res[1], 1))

        # compute sdf
        sdf = sdf_fn(x=mesh_x, y=mesh_y)

        # compute velocity grad
        velocity_grad = velocity_grad_fn(Variables({'x': mesh_x, 'y': mesh_y}))

        # compute probability density (zero if sdf is negative/outside domain)
        # probabiliy equal to the norm of the gradient
        dx = (channel_length[1] - channel_length[0]) / (grid_res[0]-1)
        dy = (2*outer_cylinder_radius) / (grid_res[1]-1)
        element_size = dx*dy
        vel_grad_norm = (velocity_grad['u__x']**2
                       + velocity_grad['u__y']**2
                       + velocity_grad['v__x']**2
                       + velocity_grad['v__y']**2)**0.5
        if self.sampled_counter <  self.importance_prob_warmup:
          unnormalized_prob = np.heaviside(sdf+max(dx,dy), 0.0) * (np.ones_like(sdf)) # uniform distribution
        else:
          unnormalized_prob = np.heaviside(sdf+max(dx,dy), 0.0) * (vel_grad_norm + self.epsilon_uniform)
        unnormalized_prob = np.reshape(unnormalized_prob, grid_res)

        # normalize probability distribution, (||prob|| = 1)
        area = (channel_length[1] - channel_length[0]) * (2*outer_cylinder_radius)
        normalized_prob = unnormalized_prob / np.sum(area * unnormalized_prob/ (grid_res[0]*grid_res[1]))

        # plot function for viewing
        plt.imshow(normalized_prob)
        plt.title("Sample Probability")
        plt.colorbar()
        plt.savefig("sample_prob.png")
        plt.close()

        return normalized_prob

      def invar_names(self):
        return ['x', 'y', 'area']

      def outvar_names(self):
        return ['continuity', 'momentum_x', 'momentum_y']

      def lambda_names(self):
        return ['lambda_continuity', 'lambda_momentum_x', 'lambda_momentum_y']

      def invar_fn(self, batch_size):
        # generate updated probability
```

```
        if self.sampled_counter == 0 or (self.sampled_counter % self.update_importance_prob_freq == 0 and self.
    sampled_counter > self.importance_prob_warmup):
          self.importance_prob_grid = self.generate_importance_prob_grid()
        self.sampled_counter += 1

        # sample points
        invar = {'x': np.zeros((0,1)),
                 'y': np.zeros((0,1)),
                 'area': np.zeros((0,1))}
        while True:
          # sample size (try to sample more then the batch so only do 1 loop)
          sample_size = 4*batch_size

          # sample points in range
          x = np.random.uniform(channel_length[0], channel_length[1], [sample_size, 1])
          y = np.random.uniform(-outer_cylinder_radius, outer_cylinder_radius, [sample_size, 1])
          rand = np.random.uniform(0, np.max(self.importance_prob_grid), [sample_size, 1])

          # compute index for interpolation
          dx = (channel_length[1] - channel_length[0]) / (grid_res[0]-1)
          dy = (2*outer_cylinder_radius) / (grid_res[1]-1)
          x_index = np.round((x - channel_length[0]) / dx).astype(dtype=np.int)
          y_index = np.round((y + outer_cylinder_radius) / dy).astype(dtype=np.int)

          # index importance grid
          prob = self.importance_prob_grid[x_index[:,0], y_index[:,0]]

          # remove points according to the rejection method (https://web.mit.edu/urban_or_book/www/book/
    chapter7/7.1.3.html)
          # also remove any point on edges with negative sdf
          sdf = sdf_fn(x=x, y=y)
          remove_criteria = np.logical_and(rand[:,0] < prob, sdf[:,0] > 0)
          x = x[remove_criteria, :]
          y = y[remove_criteria, :]
          prob = np.expand_dims(prob[remove_criteria], axis=-1)

          # add points to out dictionary
          invar['x'] = np.concatenate([invar['x'], x], axis=0)
          invar['y'] = np.concatenate([invar['y'], y], axis=0)
          invar['area'] = np.concatenate([invar['area'], prob], axis=0)

          # check if sampled enough points
          if invar['x'].shape[0] >= batch_size:
            invar['x'] = invar['x'][:batch_size]
            invar['y'] = invar['y'][:batch_size]
            invar['area'] = 1.0/(invar['area'][:batch_size]*batch_size)
            break

        return invar

      def outvar_fn(self, invar):
        outvar = {'continuity': np.zeros_like(invar['x']),
                  'momentum_x': np.zeros_like(invar['x']),
                  'momentum_y': np.zeros_like(invar['x'])}
        return outvar

      def lambda_fn(self, invar, outvar):
        lambda_weighting = {'lambda_continuity': sdf_fn(x=invar['x'], y=invar['y']) * invar['area'],
                            'lambda_momentum_x': sdf_fn(x=invar['x'], y=invar['y']) * invar['area'],
                            'lambda_momentum_y': sdf_fn(x=invar['x'], y=invar['y']) * invar['area']}
        return lambda_weighting

    # interior
    interior = ImportanceSampledInterior()
    self.add(interior, name="Interior")

    # make integral continuity
    outlet = geo.boundary_bc(outvar_sympy={'integral_continuity': 2},
                             batch_size_per_area=128,
                             lambda_sympy={'lambda_integral_continuity': 0.1},
                             criteria=Eq(x, channel_length[1]))
    self.add(outlet, name="OutletContinuity")
```

Listing 102: Defining a training domain with importance sampling.

12.8 Quasirandom Sampling

By default, SimNet generates the training points using a uniform distribution. Alternatively, one can use the Halton low-discrepancy sequence generator in SimNet (as described in Section 1.5.7). The following shows how to generate training points using a Halton sequence generator for the lid-driven cavity example by setting `quasirandom=True`.

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

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

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

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

Listing 103: Using Halton sequences to generate quasirandom training points for the lid-driven cavity example.

Please note that the Halton sequence generator in it's current form in SimNet should not be used for a geometry when specifying `fixed_var=False` for that geometry.

12.9 Example: Radial Basis Neural Networks

In this example we see the code for implementation of the RBF network and compare its results against the default Fully Connected network in SimNet.

The code for the RBF network can be found in the source code in `simnet/architecture/radial_basis.py`. Any new neural network architecture can be defined using the classes and modules used in the architectures defined in this directory. You can use any one of the pre-coded architecture as a template and make the required changes to define a new neural network. The code for the RBF neural network is shown below.

```
import tensorflow as tf
from ..config import str2bool
from ..arch import Arch
from ..variables import Variables
from ..tf_utils.activation_functions import set_nonlinearity
from ..tf_utils.layers import fc_layer, _variable

class RadialBasisArch(Arch):
  def __init__(self, **config):
    super(RadialBasisArch, self).__init__(**config)
    needed_config = RadialBasisArch.process_config(config)
    self.__dict__.update(needed_config)
    self.print_configs()
    self.bounds = None

  def set_bounds(self, bounds):
    self.bounds = bounds

  @classmethod
  def add_options(cls, group):
    group.add_argument('--nr_centers', help='number of radial basis', type=int,
                       default=128)
    group.add_argument('--sigma', help='sigma in kernel', type=float,
```

```
                          default=0.1)

  def _network_template(self, invars, out_names):
    # get input variables into tensor of shape [None, n]
    inputs = Variables.to_tensor(invars)

    # make center weight matrix
    assert self.bounds is not None, "Need to set bounds for raidal basis network"
    centers = []
    for dim in self.bounds:
      initializer = tf.initializers.random_uniform(minval=self.bounds[dim][0], maxval=self.bounds[dim][1])
      c = _variable('c_'+dim,
                    shape=[1,self.nr_centers,1],
                    initializer=initializer)
      centers.append(c)
    centers = tf.concat(centers, axis=-1)
    centers = tf.stop_gradient(centers)

    # tile centers and inputs so each tile runs on each input
    centers = tf.tile(centers,[tf.shape(inputs)[0],1,1])
    inputs = tf.expand_dims(inputs, axis=1)
    inputs = tf.tile(inputs,[1,tf.shape(centers)[1],1])

    # calc kernels
    #c = (1.0/(self.sigma*np.sqrt(2*np.pi)))
    radial_activation = tf.math.exp(-0.5*(tf.norm(centers-inputs, axis=-1)/self.sigma)**2)

    # perceptron out
    x = fc_layer(radial_activation, len(out_names), activation_fn=None, name='fc_final') # no weight norm on
     last layer
    return Variables.from_tensor(x, out_names)
```

Listing 104: Radial Basis Neural Network

We solve the Hemholtz equation in 2D defined in equation 109 [10].

$$\begin{aligned} \Delta u(x,y) + u(x,y) = q(x,y), (x,y) \in \Omega := (-1,1) \\ u(x,y) = 0, (x,y) \in \partial\Omega \end{aligned} \tag{109}$$

where,

$$q = -(\pi)^2 \sin(\pi x)\sin(4\pi y) - (4\pi)^2 \sin(\pi x)\sin(4\pi y) + \sin(\pi x)\sin(4\pi y) \tag{110}$$

The analytical solution to the above problem is given by equation 111

$$u = \sin(\pi x)\sin(4\pi y) \tag{111}$$

Note: The python script for the problem above, can be found at `examples/helmholtz/helmholtz_radial_basis.py`

We run the problem using the default fully connected architecture and the Radial Basis architecture shown above. Figure 60 shows the comparison of results between the two architectures.

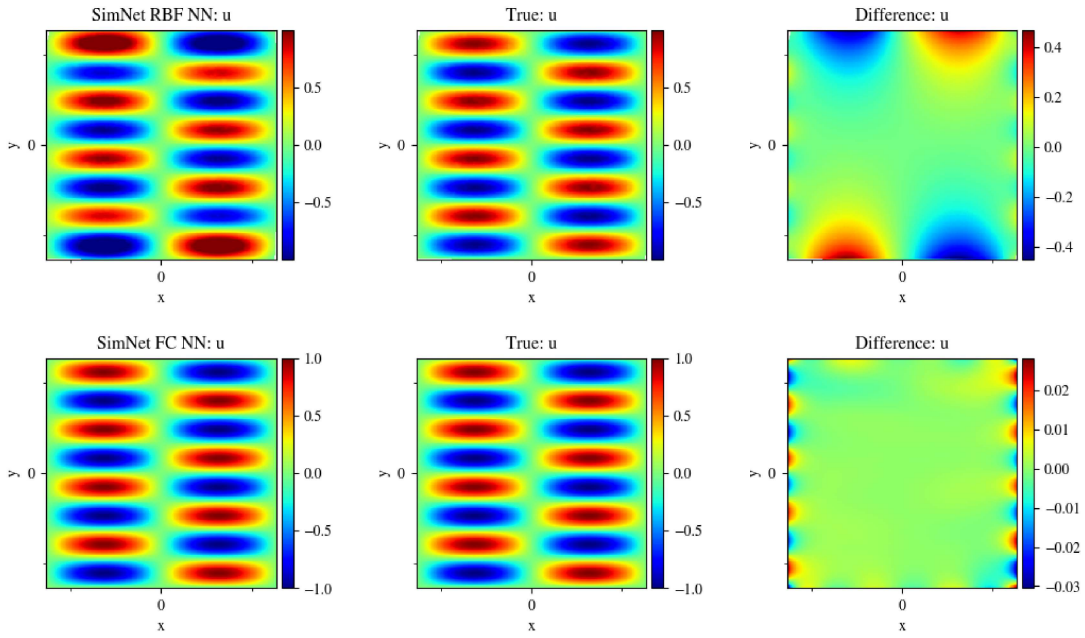


Figure 60: Radial Basis Network versus Fully connected Network

13 Multi-Physics Simulations: Conjugate Heat Transfer

13.1 Introduction

In this tutorial, we will use SimNet to study the conjugate heat transfer between the heat sink and the surrounding fluid. The temperature variations inside solid and fluid would be solved in a coupled manner with appropriate interface boundary conditions. In this tutorial, you would learn the following:

1. How to generate a 3D geometry using the geometry module in SimNet.
2. How to set up multiple point densities in different regions of the flow field.
3. How to set up a conjugate heat transfer problem using the interface boundary conditions in SimNet.
4. How to use the Muti-Phase training approach in SimNet for one-way coupled problems.

Prerequisites

This tutorial assumes that you have completed tutorial 2 on Lid Driven Cavity Flow and have familiarized yourself with the basics of the SimNet user interface. Also, we recommend you to refer to tutorial 3 for information on creating monitor and inference domains and tutorial 8 for additional details on writing some of the thermal boundary conditions.

13.2 Problem Description

The geometry for a 3-fin heat sink placed inside a channel is shown in figure 61. The inlet to the channel is at 1 m/s. The pressure at the outlet is specified as 0 Pa. All the other surfaces of the geometry are treated as no-slip walls. The dimensions of the channel and the 3-fin heat sink are shown in figure 61.

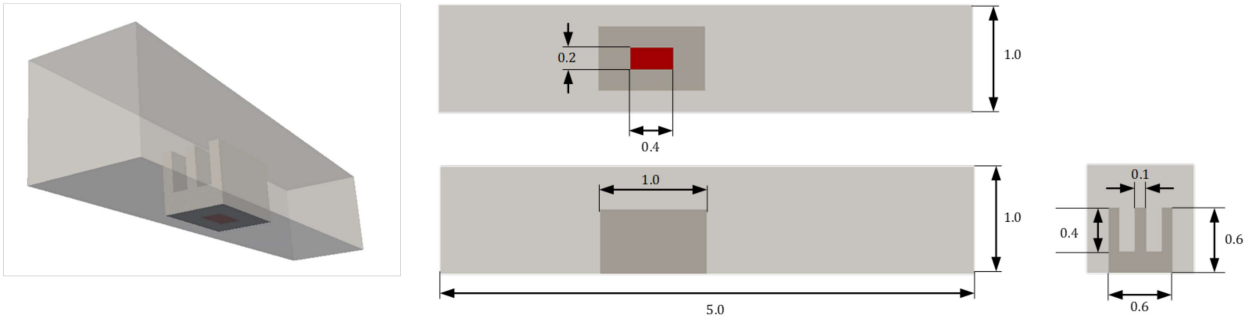


Figure 61: Three fin heat sink geometry (All dimensions in m)

The inlet is at 273.15 K. The channel walls are adiabatic. The heat sink has a heat source of 0.2x0.4 m at the bottom of the heat sink situated centrally on the bottom surface. The heat source generates heat such that the temperature gradient on the source surface is 360 K/m in the normal direction. Conjugate heat transfer takes place between the fluid-solid contact surface.

The properties fluid and thermal properties of the fluid and the solid are as follows:

Table 3: Fluid and Solid Properties

Property	Fluid	Solid
Kinematic Viscosity (m^2/s)	0.02	NA
Thermal Diffusivity (m^2/s)	0.02	0.0625
Thermal Conductivity $(W/m.K)$	1.0	5.0

13.3 Case Setup

In this tutorial, since we are dealing with only incompressible flow, there is a one-way coupling between the heat and flow equations. This means that it is possible to train the temperature field after the flow field is trained and

converged. Such an approach is useful while training the multi-physics problems which are one-way coupled as it is possible to achieve significant speed-up, as well as simulate cases with same flow boundary conditions but different thermal boundary conditions. One can easily use the same flow field as in input to train for different thermal boundary conditions.

Hence, for this problem we will have three separate files for domain, flow solver, and heat solver. The `three_fin_domain.py` will contain all the definitions of geometry and domains for flow as well as heat and the corresponding boundary conditions. `three_fin_flow_solver.py` would then take in information from the domain file and compute the flow field. Once the flow field is sufficiently converged, we can use `three_fin_heat_solver.py` to then take in the flow field generated and solve for the temperature distributions in fluid and solid simultaneously.

In this problem we will non-dimensionalize the temperature according to the following equation:

$$\theta = T/273.15 - 1.0 \tag{112}$$

Note: The python scripts for this problem can be found at `examples/three_fin_3d/laminar/`.

Importing the required packages

The list of the packages to be imported in each of the file are shown below. Only the relevant flow and heat domains/ classes are imported in `three_fin_flow_solver.py` and `three_fin_heat_solver.py` respectively.

```
from sympy import Symbol, Eq, tanh
import numpy as np
import tensorflow as tf

from simnet.dataset import TrainDomain, ValidationDomain, MonitorDomain
from simnet.data import Validation, Monitor
from simnet.sympy_utils.geometry_3d import Box, Channel, Plane
from simnet.csv_utils.csv_rw import csv_to_dict
```

Listing 105: Importing the required packages and modules for the domain file

```
# import domain
from three_fin_domain import ThreeFinFlowTrain, ThreeFinFlowMonitor, ThreeFinFlowVal, nu, rho

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

Listing 106: Importing the required packages and modules for the flow solver file

```
# import domain
from three_fin_domain import ThreeFinHeatTrain, ThreeFinHeatMonitor, ThreeFinHeatVal, D_solid, D_fluid, rho

# import SimNet library
from simnet.solver import Solver
from simnet.PDES.navier_stokes import GradNormal
from simnet.PDES.advection_diffusion import AdvectionDiffusion, IntegralAdvection
from simnet.PDES.diffusion import Diffusion, DiffusionInterface
from simnet.controller import SimNetController
```

Listing 107: Importing the required packages and modules for the heat solver file

13.3.1 Creating Geometry

As described earlier, we will edit the `three_fin_domain.py` file to create the geometry. We will use the `Box` primitive to create the 3-fin geometry and `Channel` primitive to generate the channel. Similar to 2D, `Channel` and `Box` are defined by using the two corner points. Like 2D, the `Channel` geometry has no bounding planes in the x-direction. We will also make use of the `repeat` attribute to create the fins. This attribute speeds up the generation of repetitive structures in comparison to constructing the same geometry using for/while loop.

We will use the `Plane` geometry to create the planes at the inlet and outlet. The code for generating the required geometries is shown below. Please note the normal directions for the inlet and outlet planes.

Additionally, the parameters required for solving the heat part as also defined upfront, ex. dimensions and locations of source etc.

```
# geometry params for domain
channel_origin        = (-2.5, -0.5, -0.5)
channel_dim           = (5.0, 1.0, 1.0)
heat_sink_base_origin = (-1.0, -0.5, -0.3)
heat_sink_base_dim    = (1.0, 0.2, 0.6)
fin_origin            = heat_sink_base_origin
fin_dim               = (1.0, 0.6, 0.1)
total_fins            = 3
flow_box_origin       = (-1.1, -0.5, -0.5)
flow_box_dim          = (1.2, 1.0, 1.0)
source_origin         = (-0.7, -0.5, -0.1)
source_dim            = (0.4, 0.0, 0.2)
source_area           = 0.08

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

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

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

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

# entire geometry
geo = channel - three_fin


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

Listing 108: Generating the geometry

13.3.1.1 Varying point sampling density & adding parameterized integral continuity planes

In this problem, we will sample points at a higher density in the vicinity of the heat sink. This will help in achieving a faster convergence and also higher accuracy as majority of the flow field variations are expected to occur in the vicinity of the heat sink. To achieve this, we will create an high resolution `flow_box` surrounding the heat sink which will be used to sample the points more densely. This high resolution box would be then subtracted out of the channel geometry to create the low resolution box.

We will also create some integral continuity planes in the vicinity of the heat sink to help the development of the flow field faster. Unlike tutorial 8 where we placed the planes at static location, we will parameterize them such that a new plane is drawn every training iteration. These planes will enforce the constraint of satisfying a specified mass flow through these planes. These parameterized planes would be used to set the `IntegralContinuity` condition in the equations and solver parts of the code. To generate these parameterized planes, a symbolic variable for x position of the

plane would be created and we will randomly sample planes in the range (-1.1, 0.1) which is $\pm$ 0.1 m in the aft and rear of the heat sink in x direction. In this section we will only define the planes in a symbolic fashion and specify the ranges for the `x_pos`. Actual sampling of the planes would be done during the boundary condition definition.

The code to generate a separate `flow_box` and parameterized `integral_plane` can be found below.

Note: The addition of integral continuity planes and separate flow box for dense sampling are optional. However, they have been found to improve the accuracy and convergence to a great extent and it is recommended to enforce such techniques for any complex flow phenomena.

```
# low and high resultion geo away and near the heat sink
flow_box = Box(flow_box_origin, (flow_box_origin[0]+flow_box_dim[0], # base of heat sink
                                 flow_box_origin[1]+flow_box_dim[1],
                                 flow_box_origin[2]+flow_box_dim[2]))
lr_geo = geo - flow_box
hr_geo = geo & flow_box
lr_bounds_x = (channel_origin[0], channel_origin[0] + channel_dim[0])
lr_bounds_y = (channel_origin[1], channel_origin[1] + channel_dim[1])
lr_bounds_z = (channel_origin[2], channel_origin[2] + channel_dim[2])
hr_bounds_x = (flow_box_origin[0], flow_box_origin[0] + flow_box_dim[0])
hr_bounds_y = (flow_box_origin[1], flow_box_origin[1] + flow_box_dim[1])
hr_bounds_z = (flow_box_origin[2], flow_box_origin[2] + flow_box_dim[2])

# planes for integral continuity
x_pos = Symbol('x_pos')
integral_plane = Plane((x_pos, channel_origin[1], channel_origin[2]),
                       (x_pos, channel_origin[1]+channel_dim[1], channel_origin[2]+channel_dim[2]),
                       1)
x_pos_range = {x_pos: lambda batch_size: np.full((batch_size, 1), np.random.uniform(-1.1, 0.1))}
```

Listing 109: Adding geometries for dense point sampling and integral continuity planes

13.3.2 Defining the Flow Boundary conditions and Equations

For the multi-phase training approach, we will generate the `ThreeFinFlowTrain` containing all the flow boundary conditions and `ThreeFinHeatTrain` containing all the thermal boundary conditions separately.

The contents of `ThreeFinFlowTrain` are listed below.

Inlet, Outlet and Channel and Heat Sink walls For inlet boundary conditions, we will specify the velocity to be a constant velocity of 1.0 m/s in x-direction. Like in tutorial 2, we will weight the velocity by the SDF of inlet plane to avoid sharp discontinuity at the boundaries. For outlet, we will specify the pressure to be 0. All the channel walls and heat sink walls are treated as no slip boundaries.

Interior The flow equations can be specified in the low resolution and high resolution domains of the problem by using `interior_bc`. We will use different `batch_size_per_area` for each of these domains (`interiorF_lr` and `interiorF_hr`) to vary the density of points.

Integral Continuity The integral continuity planes would be sampled by using a for loop as shown in the code. The inlet volumetric flow is 1 m^3/s and hence we will specify `integral_continuity` variable as 1.0 on each of these planes.

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

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

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

    # flow interior low res away from three fin
    interiorF_lr = lr_geo.interior_bc(outvar_sympy={'continuity': 0, 'momentum_x': 0, 'momentum_y': 0, '
     momentum_z': 0},
                                      bounds={x: lr_bounds_x, y: lr_bounds_y, z: lr_bounds_z},
                                      lambda_sympy={'lambda_continuity': geo.sdf,
                                                    'lambda_momentum_x': geo.sdf,
                                                    'lambda_momentum_y': geo.sdf,
                                                    'lambda_momentum_z': geo.sdf},
                                      batch_size_per_area=500)
    self.add(interiorF_lr, name="FlowInterior_LR")

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

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

Listing 110: Defining the flow boundary conditions and equations

13.3.3 Defining the Thermal Multi-Phase Boundary conditions and Equations

The contents of `ThreeFinHeatTrain` are described below.

Inlet, Outlet and Channel walls: For the heat part, we will specify temperature at the inlet. All the outlet and the channel walls will have a zero gradient boundary condition which will be enforced by setting `'normal_gradient_theta_f'` equal to 0. We will use `'theta_f'` for defining the temperatures in fluid and `'theta_s'` for defining the temperatures in solid.

Fluid and Solid Interior: Just like tutorial 8, we will set `'advection_diffusion'` equal to 0 in both, low and high resolution fluid domains. For solid interior, we will set `'diffusion'` equal to 0.

Fluid-Solid Interface: For the interface between fluid and solid, we will enforce both Neumann and Dirchlet boundary condition by setting `'diffusion_interface_dirichlet_theta_f_theta_s'` and `'diffusion_interface_neumann_theta_f_theta_s'` both equal to 0.

Note: The order in which you define `'theta_f'` and `'theta_s'` in the interface boundary condition is important. The different conductivities for fluid and solid will be specified in the same order during the heat solver file definition.

Heat Source: We will apply a $tanh$ smoothing while defining the heat source on the bottom wall of the heat sink. We have found that smoothing out the sharp boundaries helps in training the Neural Network converge faster. The `'normal_gradient_theta_s'` is set equal to `grad_t` in the area of source and 0 everywhere else on the bottom surface of heat sink.

Integral Advection: In addition to the regular boundary conditions, like integral continuity, we will impose the conservation of heat flux in an integral form on the outlet plane using `'integral_advection'` keyword. The heat flux out of the system must be equal to the influx which is the total of flux entering through the inlet and the source. For

this problem, applying such balance give us the following:

$$\int \rho_{fluid} c_{fluid} T_{out} U_{out} ds = \rho_{fluid} c_{fluid} T_{in} A_{in} U_{in} + k_{solid}(dT/dn) A_{source} \tag{113}$$

$$\int T_{out} U_{out} ds = T_{in} A_{in} U_{in} + \frac{k_{solid}(dT/dn) A_{source}}{\rho_{fluid} c_{fluid}} \tag{114}$$

```
class ThreeFinHeatTrain(TrainDomain):
  def __init__(self, **config):
    super(ThreeFinHeatTrain, self).__init__()
    # inlet
    inletBC = inlet.boundary_bc(outvar_sympy={'theta_f': inlet_t},
                                batch_size_per_area=500,
                                criteria=Eq(x, channel_origin[0]))
    self.add(inletBC, name="Inlet")

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

    # channel walls insulating
    walls = channel.boundary_bc(outvar_sympy={'normal_gradient_theta_f': 0},
                                batch_size_per_area=500,
                                criteria=three_fin.sdf<-1e-5)
    self.add(walls, name="ChannelWalls")

    # flow interior low res away from three fin
    interiorF_lr = lr_geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                      bounds={x: lr_bounds_x, y: lr_bounds_y, z: lr_bounds_z},
                                      batch_size_per_area=500)
    self.add(interiorF_lr, name="FlowInterior_LR")

    # flow interiror high res near three fin
    interiorF_hr = hr_geo.interior_bc(outvar_sympy={'advection_diffusion': 0},
                                      bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                      batch_size_per_area=3000)
    self.add(interiorF_hr, name="FlowInterior_HR")

    # solid interior
    interiorS = three_fin.interior_bc(outvar_sympy={'diffusion_theta_s': 0},
                                  bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z},
                                  batch_size_per_area=30000,
                                  lambda_sympy={'lambda_diffusion_theta_s': 100})
    self.add(interiorS, name="SolidInterior")

    # fluid solid interface
    interface = three_fin.boundary_bc(outvar_sympy={'diffusion_interface_dirichlet_theta_f_theta_s': 0, '
     diffusion_interface_neumann_theta_f_theta_s': 0},
                                  batch_size_per_area=500,
                                  criteria=channel.sdf>0)
    self.add(interface, name="Interface")

    # heat source
    sharpen_tanh = 60.0
    source_func_xl = (tanh(sharpen_tanh*(x - source_origin[0])) + 1.0)/2.0
    source_func_xh = (tanh(sharpen_tanh*((source_origin[0]+source_dim[0]) - x)) + 1.0)/2.0
    source_func_zl = (tanh(sharpen_tanh*(z - source_origin[2])) + 1.0)/2.0
    source_func_zh = (tanh(sharpen_tanh*((source_origin[2]+source_dim[2]) - z)) + 1.0)/2.0
    gradient_normal = grad_t * source_func_xl * source_func_xh * source_func_zl * source_func_zh
    heat_source = three_fin.boundary_bc(outvar_sympy={'normal_gradient_theta_s': gradient_normal},
                                    batch_size_per_area=5000,
                                    criteria=Eq(y, source_origin[1]))
    self.add(heat_source, name="HeatSource")

    # integral advection diffusion
    IC = outlet.boundary_bc(outvar_sympy={'integral_advection_theta_f': 5.0*source_area*grad_t/50 + inlet_t*1.0*
     inlet_vel},
                            lambda_sympy={'lambda_integral_advection_theta_f': 0.1},
                            batch_size_per_area=10000)
    self.add(IC, name="IntegralAdvection")
```

Listing 111: Defining the thermal boundary conditions and equations

13.3.4 Creating Validation and Monitor domains

Like separate training domains, we will create separate validation, and monitor domains for flow and heat part. In this tutorial we will monitor the residuals of flow equations and pressure drops during the flow field simulation and monitor the residuals of heat equations and peak temperature reached in the source chip during the heat simulation in the second phase.

```
# validation data
# flow data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z',
           'U:0': 'u', 'U:1': 'v','U:2': 'w', 'p_rgh': 'p', 'T': 'theta_f'}
openfoam_var = csv_to_dict('../openfoam/threeFin_extend_fluid0.csv', mapping)
openfoam_var['theta_f'] = openfoam_var['theta_f']/273.15 - 1.0 # normalize heat
openfoam_var['x'] = openfoam_var['x'] + channel_origin[0]
openfoam_var['y'] = openfoam_var['y'] + channel_origin[1]
openfoam_var['z'] = openfoam_var['z'] + channel_origin[2]
openfoam_invar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['x','y','z']}
openfoam_outvar_flow_numpy = {key: value for key, value in openfoam_var.items() if key in ['u','v','w','p']}
openfoam_outvar_flow_heat_numpy = {key: value for key, value in openfoam_var.items() if key in ['theta_f']}
# solid data
mapping = {'Points:0': 'x', 'Points:1': 'y', 'Points:2': 'z',
           'T': 'theta_s'}
openfoam_var = csv_to_dict('../openfoam/threeFin_extend_solid0.csv', mapping)
openfoam_var['theta_s'] = openfoam_var['theta_s']/273.15 - 1.0 # normalize heat
openfoam_var['x'] = openfoam_var['x'] + channel_origin[0]
openfoam_var['y'] = openfoam_var['y'] + channel_origin[1]
openfoam_var['z'] = openfoam_var['z'] + channel_origin[2]
openfoam_invar_solid_numpy = {key: value for key, value in openfoam_var.items() if key in ['x','y','z']}
openfoam_outvar_solid_numpy = {key: value for key, value in openfoam_var.items() if key in ['theta_s']}

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

class ThreeFinHeatVal(ValidationDomain):
  def __init__(self, **config):
    super(ThreeFinHeatVal, self).__init__()
    # heat in fluid validation
    val = Validation.from_numpy(openfoam_invar_flow_numpy, openfoam_outvar_flow_heat_numpy)
    self.add(val, name='ValFlowHeat')

    # heat in solid validation
    val = Validation.from_numpy(openfoam_invar_solid_numpy, openfoam_outvar_solid_numpy)
    self.add(val, name='ValSolidHeat')
```

Listing 112: Defining the Validation domain

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

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

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

class ThreeFinHeatMonitor(MonitorDomain):
  def __init__(self, **config):
```

```
    super(ThreeFinHeatMonitor, self).__init__()
    # metric for equation residuals
    global_monitor = Monitor(geo.sample_interior(10000, bounds={x: hr_bounds_x, y: hr_bounds_y, z: hr_bounds_z})
    ,
                             {'advection_diffusion': lambda var: tf.reduce_sum(var['area']*tf.abs(var['
     advection_diffusion']))})
    self.add(global_monitor, 'FlowResiduallMonitor')
    global_monitor = Monitor(three_fin.sample_interior(100000, bounds={x: hr_bounds_x, y: hr_bounds_y, z:
     hr_bounds_z}),
                             {'diffusion': lambda var: tf.reduce_sum(var['area']*tf.abs(var['diffusion_theta_s'])
     )})
    self.add(global_monitor, 'SolidResiduallMonitor')

    # metric for peak temp
    temp_monitor = Monitor(three_fin.sample_boundary(10000, criteria=Eq(y, source_origin[1])),
                           {'peak_temp': lambda var: tf.reduce_max(var['theta_s'])})
    self.add(temp_monitor, 'PeakTempMonitor')
```

Listing 113: Defining the monitors

13.4 Making the neural network, Multi-Phase training

Once all the domain definitions are completed in the `three_fin_domain.py` file, we can complete the `three_fin_flow_solver.py` to add all the required equations and run the SimNet solver. This part of the code is similar to what we have seen in previous tutorials. `three_fin_flow_solver.py` can be found below.

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

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

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

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

Listing 114: Defining flow solver

Once the flow part is trained, we will use that converged checkpoint to initialize the training for the heat part. This can be done by specifying the fluid flow's network directory as an initializer for the heat simulation. To achieve this, we specify the path for fluid flow network directory to the `'initialize_network_dir'` key in `update_defaults` function.

Also, for making the multi-phase approach to work, you will need to maintain the exact same flow network in the heat solver file as well. Thus, in the heat solver file, along with creating two networks for solving the heat in the solid and fluid, we will create the same flow network as used in the flow solver file.

Relevant equations are added in a similar way as seen in the flow solver file. Please note the order of `'theta_f'` and `'theta_s'` while specifying the `DiffusionInterface` equation. The order must be same as used in train domain's definition. The following two variables are the conductivites for fluid and solid respectively. Similar to tutorial 8, we will stop the gradients for velocity components in the `AdvectionDiffusion` equation.

The code for the `three_fin_heat_solver.py` can be found below.

```
class ThreeFinHeatSolver(Solver):
  train_domain = ThreeFinHeatTrain
```

```
  monitor_domain = ThreeFinHeatMonitor
  val_domain = ThreeFinHeatVal

  def __init__(self, **config):
    super(ThreeFinHeatSolver, self).__init__(**config)
    self.equations = (AdvectionDiffusion(T='theta_f', rho=rho, D=D_fluid, dim=3, time=False).make_node(
     stop_gradients=['u', 'v', 'w'])
                      + Diffusion(T='theta_s', D=D_solid, dim=3, time=False).make_node()
                      + DiffusionInterface('theta_f', 'theta_s', 1.0, 5.0, dim=3, time=False).make_node()
                      + GradNormal('theta_f', dim=3, time=False).make_node()
                      + GradNormal('theta_s', dim=3, time=False).make_node()
                      + IntegralAdvection(T='theta_f', dim=3).make_node())
    flow_net = self.arch.make_node(name='flow_net',
                                   inputs=['x', 'y', 'z'],
                                   outputs=['u', 'v', 'w', 'p'])
    solid_net = self.arch.make_node(name='solid_net',
                                    inputs=['x', 'y', 'z'],
                                    outputs=['theta_s'])
    flow_heat_net = self.arch.make_node(name='flow_heat_net',
                                        inputs=['x', 'y', 'z'],
                                        outputs=['theta_f'])
    self.nets = [flow_net, solid_net, flow_heat_net]

  @classmethod
  def update_defaults(cls, defaults):
    defaults.update({
        'network_dir': './network_checkpoint_three_fin_heat',
        'initialize_network_dir': './network_checkpoint_three_fin_fluid_flow',
        'rec_results_cpu': True,
        'max_steps': 500000,
        'decay_steps': 5000,
        })

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

Listing 115: Defining the heat solver

13.5 Running the SimNet Solver

Once both the solver files are defined, we will run the `three_fin_flow_solver.py` first to solve for the flow field. Once a desired level of convergence is achieved, we can end the simulation and run the `three_fin_heat_solver.py` file.

13.6 Results and Post-processing

Table 4 shows the results of Pressure drop and Peak temperatures obtained from the SimNet and compares it with the results from OpenFOAM solver.

Table 4: Comparisons of Results with OpenFOAM

	SimNet	OpenFOAM
Pressure Drop (Pa)	7.51	7.49
Peak Temperature $(°C)$	78.35	78.05

13.6.1 Plotting gradient quantities: Wall Velocity Gradients

In a variety of applications, it is desirable to plot the gradients of some quantities inside the domain. One such example relevant to fluid flows is the wall velocity gradients and wall shear stresses. These quantities are often plotted to compute frictional forces, etc. We can visualize such quantities in SimNet by outputting the x, y and z derivatives of the desired variables using an `Inference` domain. Below we show an example of how to create such a inference domain for outputting velocity gradients on the walls of the domain.

```
class ThreeFinFlowInference(InferenceDomain):
  def __init__(self, **config):
    super(ThreeFinFlowInference, self).__init__()
    # Inference domain for wall velocity gradients
```

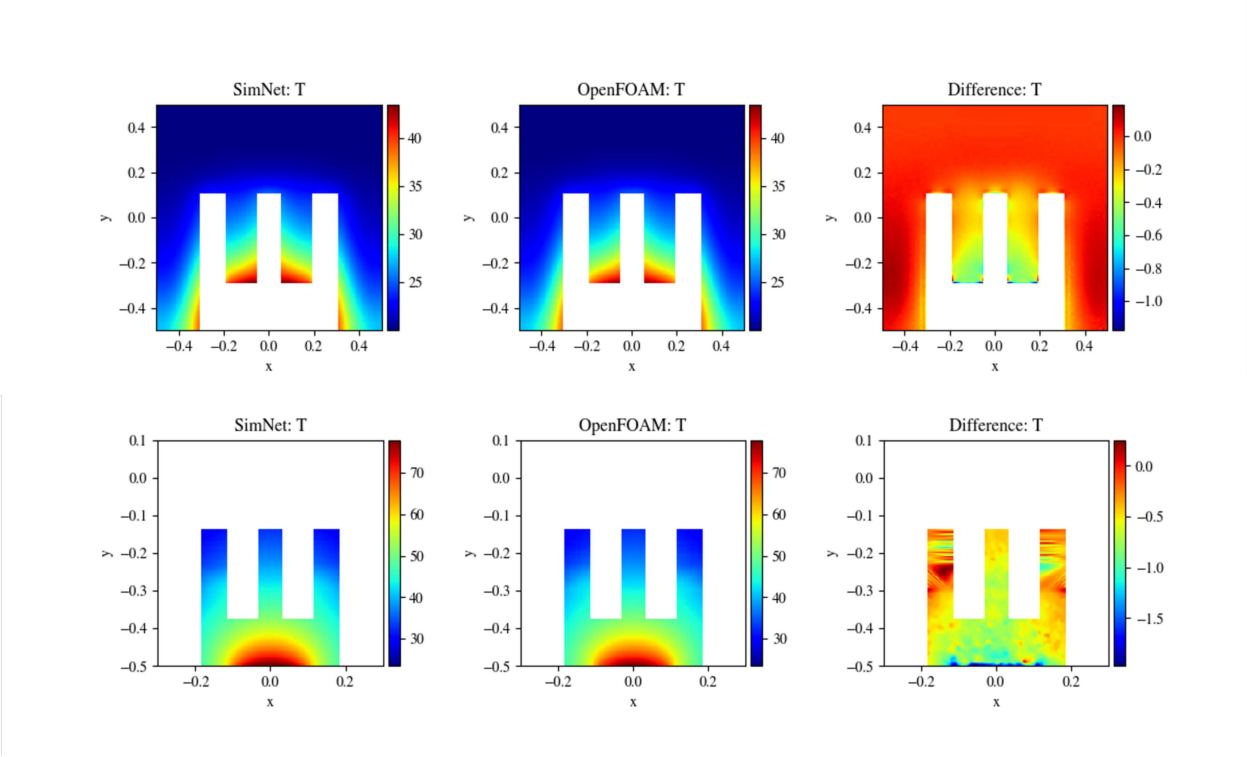


Figure 62: Left: SimNet. Center: OpenFOAM. Right: Difference. Top: Temperature distribution in Fluid. Bottom: Temperature distribution in Solid (*Temperature scales in °C*)

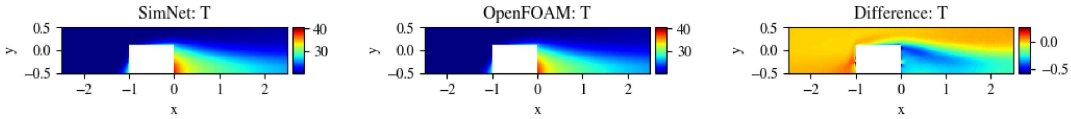


Figure 63: Left: SimNet. Center: OpenFOAM. Right: Difference. (*Temperature scales in °C*)

```
    walls = Inference(geo.sample_boundary(100000), ['u__x', 'u__y', 'u__z',
                                                    'v__x', 'v__y', 'v__z',
                                                    'w__x', 'w__y', 'w__z'])

    self.add(walls, name='WallGradients')
```

Listing 116: Post-processing gradient quantities

As we have seen, such domains can also be added as a post-processing step, after training and then the script can be executed with `--run_mode=eval` flag to just run the inference.

Then, we can post-process these quantities based on our choice to visualize the desired variables using Paraview's Calculator Filter https://kitware.github.io/paraview-docs/latest/python/paraview.simple.Calculator.html. The wall velocity gradients comparison between OpenFOAM and SimNet is shown in Figure 64.

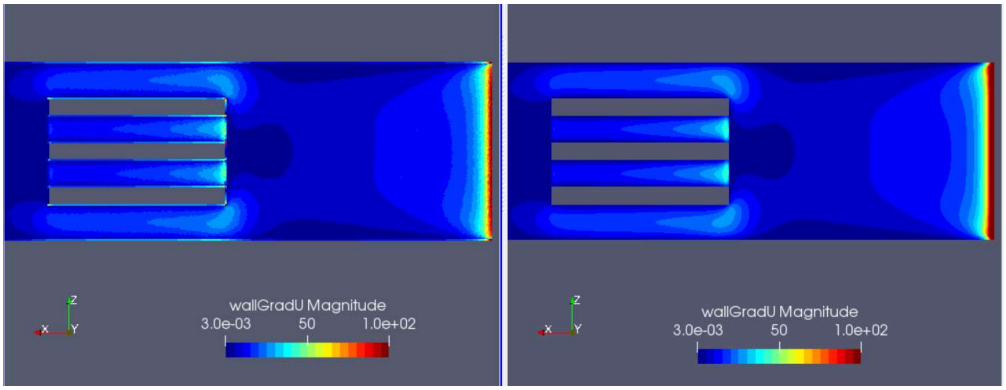


Figure 64: Comparison of magnitude of wall velocity gradients. Left: SimNet. Right: OpenFOAM