OpenVFOAM®

OpenFOAM Machine Learning Hackathon

physics-based-dl-solution-team-03 and 04

Team 3 Members

Ryley McConkey
Junsu Shin
Reza Lotfi

Team 4 Members

Rahul Sundar
Abhijeet Vishwarao
Biniyam Sishah

Supervisors

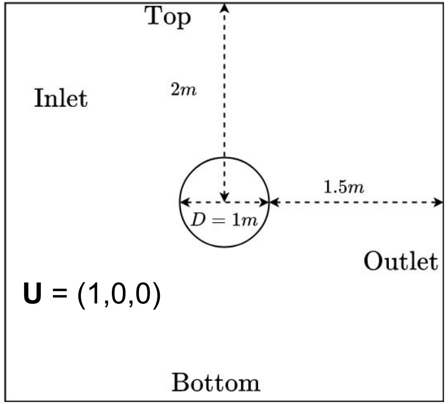
Tomislav Maric
Andre Weiner

Objective

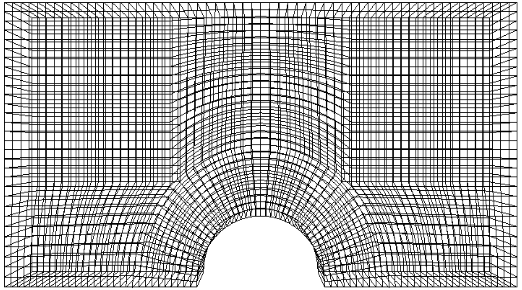
"To implement **Physics Informed Neural Networks (PINNs)** using **pytorch C++ API and** integrate with **OpenFOAM** for inferring flow fields of potential flow past a stationary cylinder"

Potential flow case setup

Schematic of 2D potential flow past a stationary cylinder

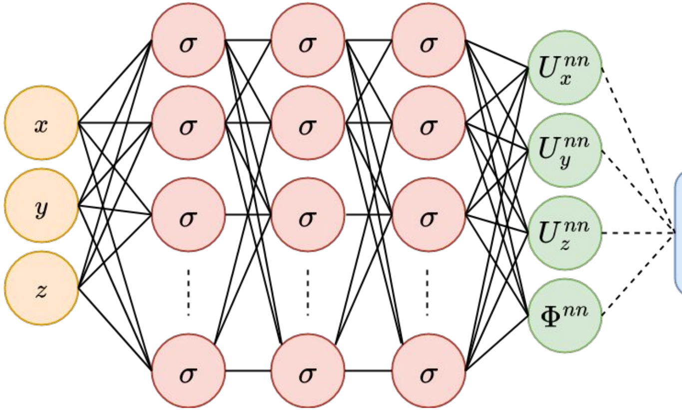


Computational domain used in potentialFoam test case



Owing to symmetry, only half domain is chosen

Pure data driven MLP: DNN



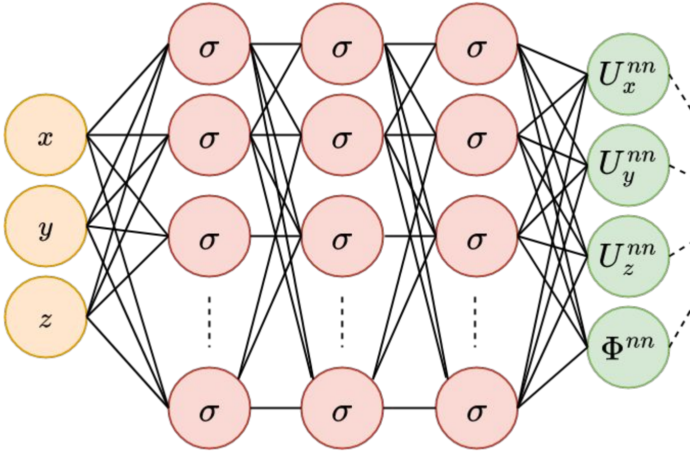
$$MSE_{data} := \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \left(\|U_x^{nn} - U_x\| + \|U_y^{nn} - U_y\| + \|U_z^{nn} - U_z\| + \|\Phi^{nn} - \Phi\| \right)$$

<u>Fixed hyperparameters</u>

Optimiser - RMSProp

Activation - GELU

Physics driven MLP: PINN



$$MSE_{data} := \frac{1}{N_{train}} \sum_{i=1}^{N_{train}} \left(\|U_x^{nn} - U_x\| + \|U_y^{nn} - U_y\| + \|U_z^{nn} - U_z\| + \|\Phi^{nn} - \Phi\| \right)$$

$$MSE_{residual} := \nabla . \nabla \Phi^{nn} - \nabla . \boldsymbol{U}^{nn}$$

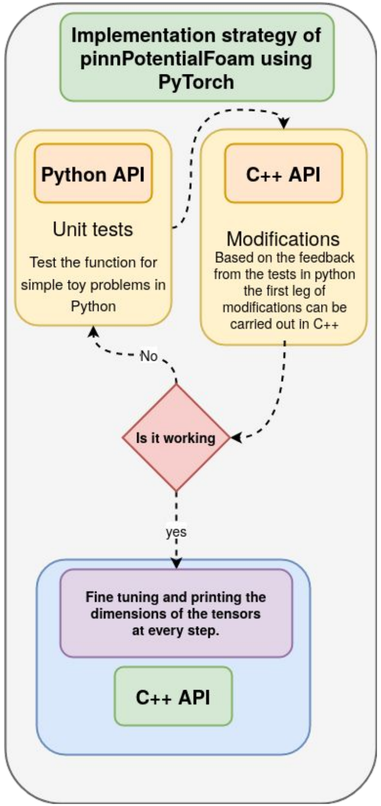
$$MSE_{total} = MSE_{data} + MSE_{residual}$$

<u>Fixed hyperparameters:</u>

Optimiser - RMSProp

Activation - GELU

Implementation strategy using libtorch



Challenges faced and strategies in place:

- Libtorch doesn't have an complete documentation
- Many standard functions of pytorch python API not available in C++ such as tile, repeat, etc.
- Component wise differentiation in an issue.
- Converting all the vectors to individual scalars and then taking their gradients helps.
- Gradients of higher order need to be computed with care.
- Unit tests in python and dimension sanity checks in C++ are a must without which the gradients computed can't be trusted.
- Gradient computation not straight forward for vector to vector or tensor to tensor/vector jacobians.

Project folder Structure

physics-based-dl-team-solution-03-4/
---applications/
 ---pinnFoam/
 ---pinnFoamSetSphere/
 ---dnnPotentialFoam/
 ---pinnPotentialFoam/
---run/
 ---dnnCylinder/
 ---pinnCylinder/
 ---dnnCylinderHOPT_grid/
 ---dnnCylinderHOPT_bayes/
 ---pinnCylinderHOPT_grid/
 ---pinnCylinderHOPT_bayes/

CONTD……

CONTD…..

physics-based-dl-team-solution-03-4/
---schematics/
---plots/
---README.md
---Allrun
---Allmake
---Allclean
---Presentation.pdf
---ProjectReport.pdf

Overall code changes

Standard MLP based flow field inference solver: **dnnPotentialFoam**

1. **Changes in createFields.h to:**
   a. read Phi and U from potentialFoam results
   b. Write Phi_nn, U_nn and error_nn predicted from the network.
2. **Changes in pinnFoam.C to dnnPotentialFoam:**
   a. Switch off the gradient terms
   b. Convert th scalar field to a vector field ‘O’ with 4 components
3. Finally make modifications in files and options to **dnnPotentialFoam**

PINN based flow field inference solver: **pinnPotentialFoam**

1. **Same as step 1 in the left box**
2. **Changes in pinnFoam.C to pinnPotentialFoam:**
   a. Convert th scalar field to a vector field ‘O’ with 4 components
   b. Switch on the gradient terms
   c. Calculate divergence and laplace operators (seen in the next slides)
   d. Compute residual loss and add it to the total loss.
3. Finally make modifications in files and options to **pinnPotentialFoam**

OpenFOAM®

Implementing divergence operator

```
//grad(Ux) = gradient of scalar component Ux w.r.t (x,y,z)
auto Ux_predict_grad = torch::autograd::grad(
    {O_predict.index({Slice(),0})},//N_{train} x 1
    {cc_training}, // N_{train} x 3
    {torch::ones_like(O_training.index({Slice(),0}))}, // N_{train} x 1
    true,
    true
);

//grad(Uy) = gradient of scalar component Uy w.r.t (x,y,z)
auto Uy_predict_grad = torch::autograd::grad(
    {O_predict.index({Slice(),1})},//N_{train} x 1
    {cc_training}, // N_{train} x 3
    {torch::ones_like(O_training.index({Slice(),1}))}, // N_{train} x 1
    true,
    true
);

//grad(Uz) = gradient of scalar component Uz w.r.t (x,y,z)
auto Uz_predict_grad = torch::autograd::grad(
    {O_predict.index({Slice(),2})},//N_{train} x 1
    {cc_training}, // N_{train} x 3
    {torch::ones_like(O_training.index({Slice(),2}))}, // N_{train} x 1
    true,
    true
);

auto divU = Ux_predict_grad[0].index({Slice(), 0}) + Uy_predict_grad[0].index({Slice(), 1}) + Uz_predict_grad[0].index({Slice(), 2});
```

Implementing Laplacian operator


```
// grad(Phi) = gradient of the scalar potenial Phi w.r. (x,y,z)
auto Phi_predict_grad = torch::autograd::grad(
   {O_predict.index({Slice(),3})},//N_{train} x 1
   {cc_training}, // N_{train} x 3
   {torch::ones_like(O_training.index({Slice(),3}))}, // N_{train} x 1
   true,
   true
);


auto Phi_predict_grad_x_grad = torch::autograd::grad(
   {Phi_predict_grad[0].index({Slice(),0})},//N_{train} x 1
   {cc_training}, // N_{train} x 3
   {torch::ones_like(Phi_predict_grad[0].index({Slice(),0}))}, // N_{train} x 1
   true,
   true
);


 auto Phi_predict_grad_y_grad = torch::autograd::grad(
   {Phi_predict_grad[0].index({Slice(),1})},//N_{train} x 1
   {cc_training}, // N_{train} x 3
   {torch::ones_like(Phi_predict_grad[0].index({Slice(),1}))}, // N_{train} x 1
   true,
   true
);

 auto Phi_predict_grad_z_grad = torch::autograd::grad(
   {Phi_predict_grad[0].index({Slice(),2})},//N_{train} x 1
   {cc_training}, // N_{train} x 3
   {torch::ones_like(Phi_predict_grad[0].index({Slice(),2}))}, // N_{train} x 1
   true,
   true
);


auto laplacePhi = Phi_predict_grad_x_grad[0].index({Slice(), 0}) + Phi_predict_grad_y_grad[0].index({Slice(), 1}) + Phi_predict_grad_z
```

Loss formulation in pinnPotentialFoam

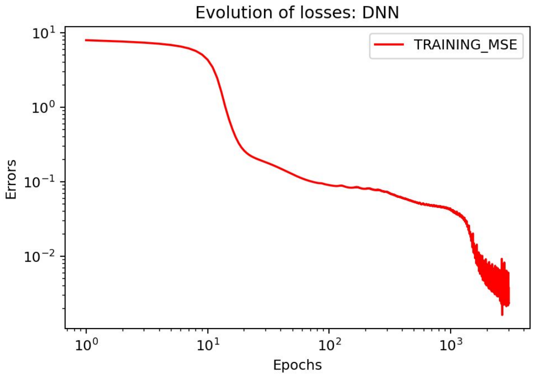
```
  auto mse_data = 4*mse_loss(O_predict, O_training);
//  );

  // div.grad(Phi) - div.U = 0

  auto potentialEqnResidual = laplacePhi - divU;

  auto mse_grad = mse_loss(
      potentialEqnResidual,
      torch::zeros_like(O_training.index({Slice(), 0}))
  );

  // Combine the losses into a Physics Informed Neural Network.
  mse = mse_data + mse_grad;
```

Note: In **dnnPotentialFoam** - **mse_grad** is omitted. And **mse = mse_data**

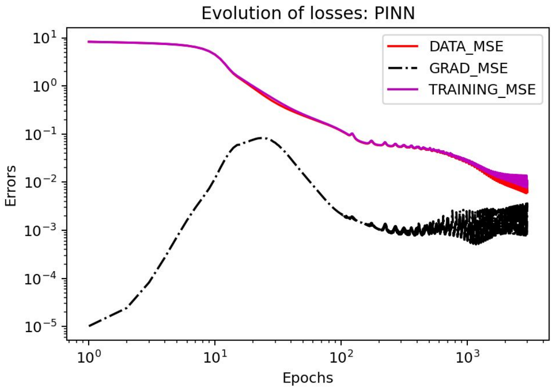
OpenFOAM®

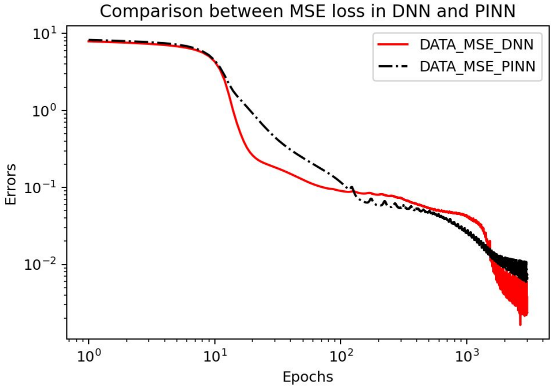
Results: Loss convergence

Case 1: Standard MLP with pure data driven loss.



Case 2: Plain vanilla PINN

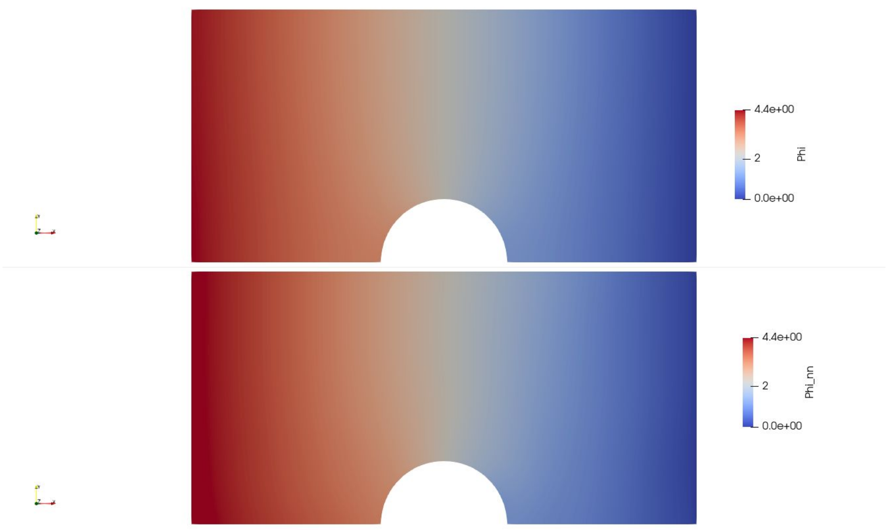




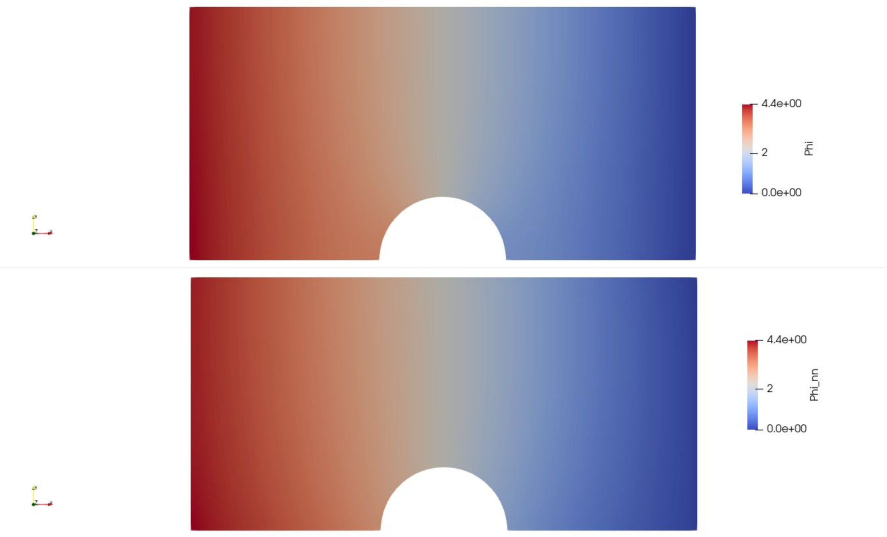
OpenFOAM®

Results: Phi

Case 1: Standard MLP with pure data driven loss.

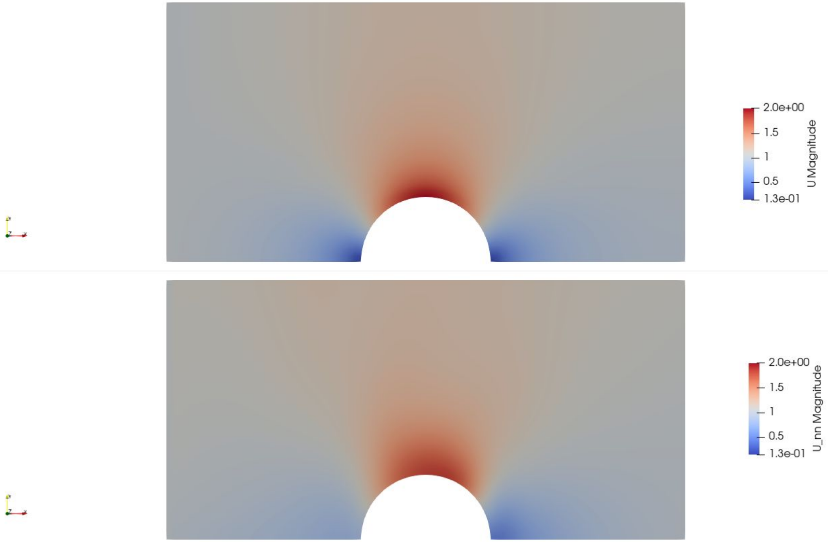


Case 2: Plain vanilla PINN

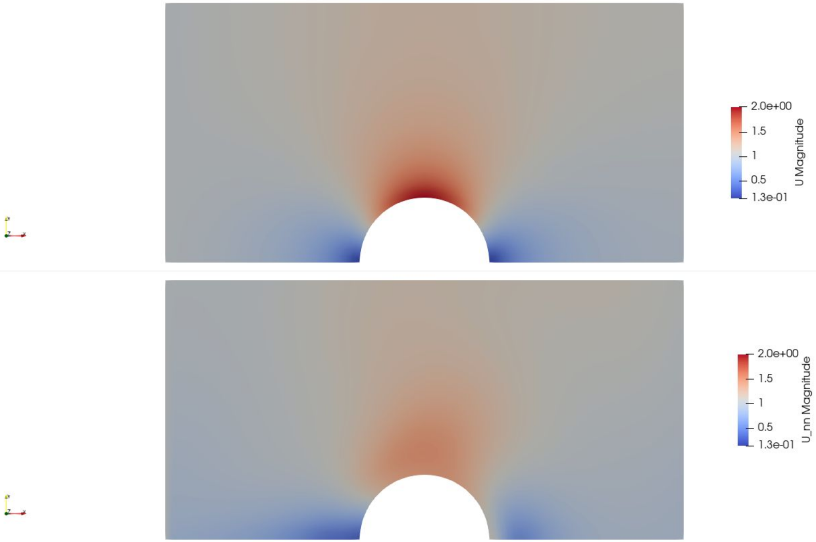


Results: U magnitude

Case 1: Standard MLP with pure data driven loss.



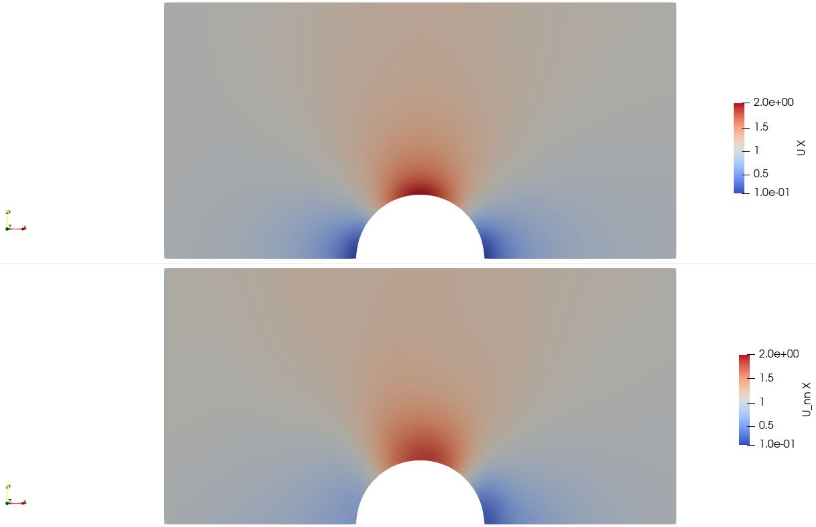
Case 2: Plain vanilla PINN



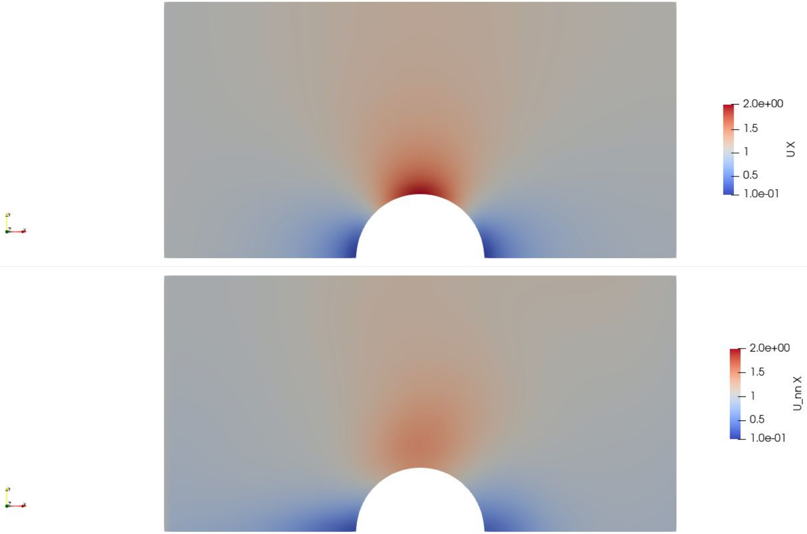
OpenVFOAM®

Results: U_x

Case 1: Standard MLP with pure data driven loss.



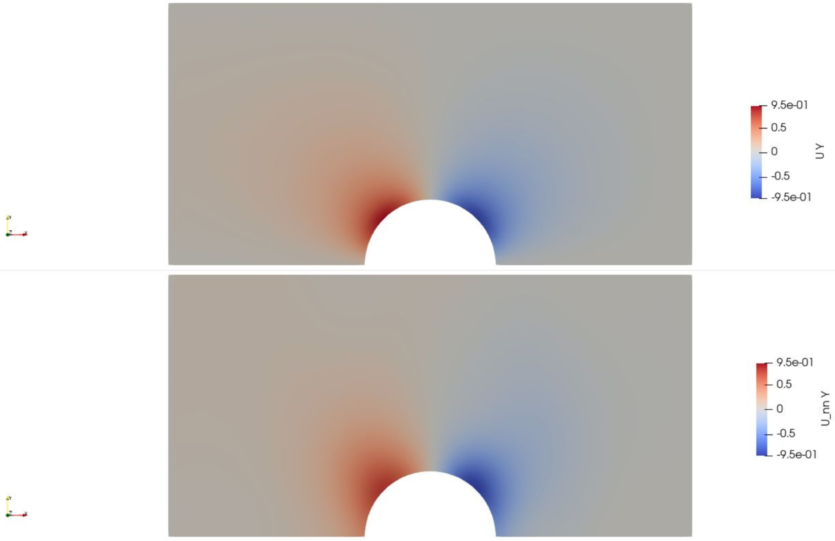
Case 2: Plain vanilla PINN



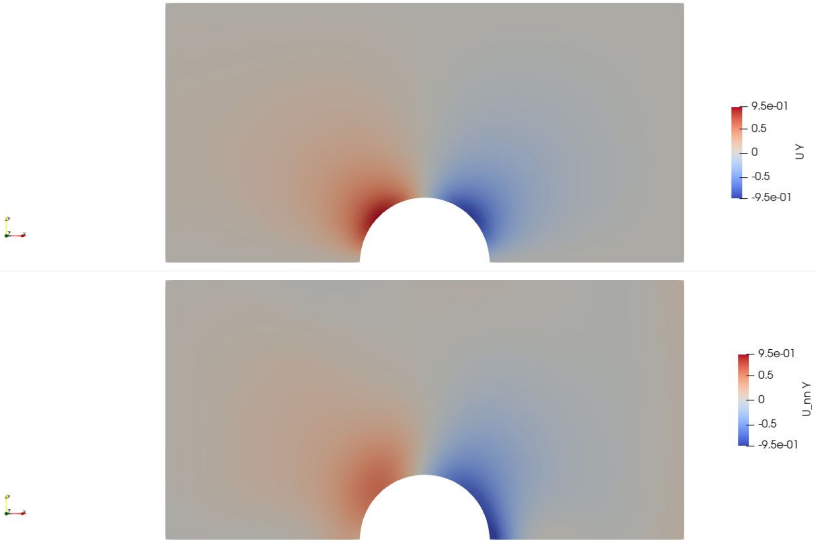
OpenFOAM®

Results: U_y

Case 1: Standard MLP with pure data driven loss.



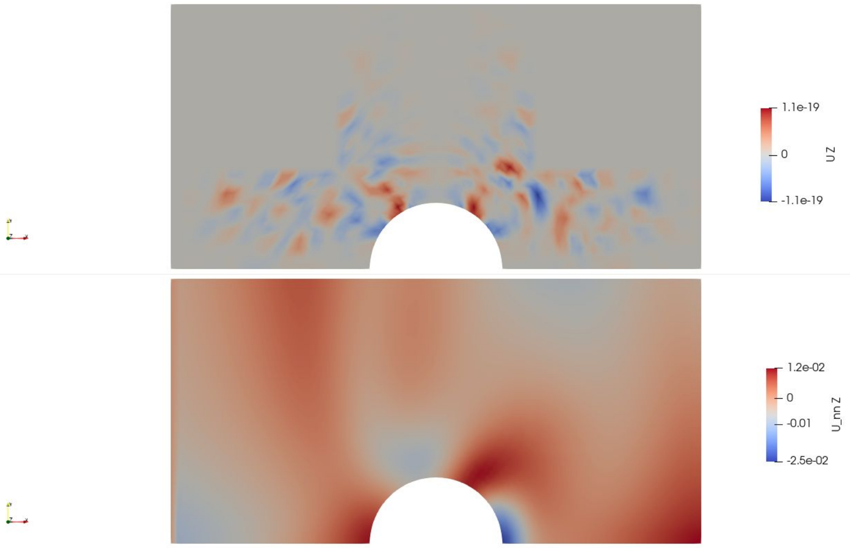
Case 2: Plain vanilla PINN



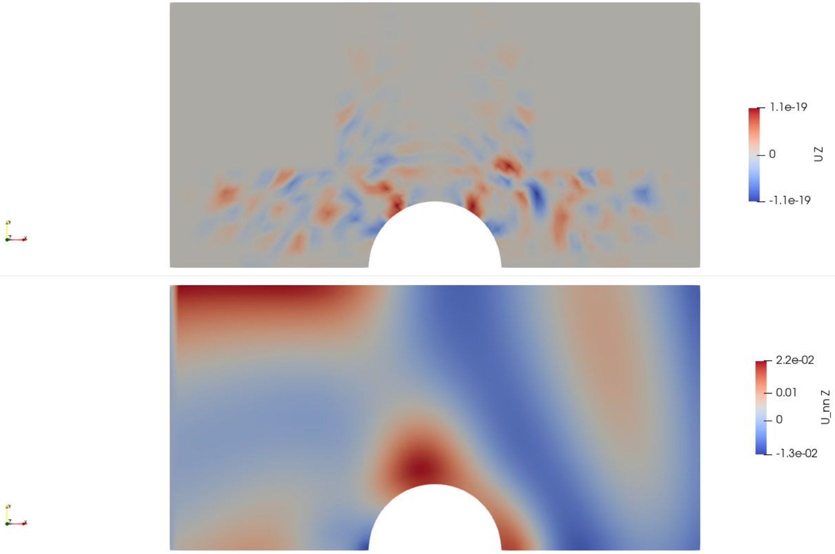
OpenFOAM®

Results: U_z

Case 1: Standard MLP with pure data driven loss.



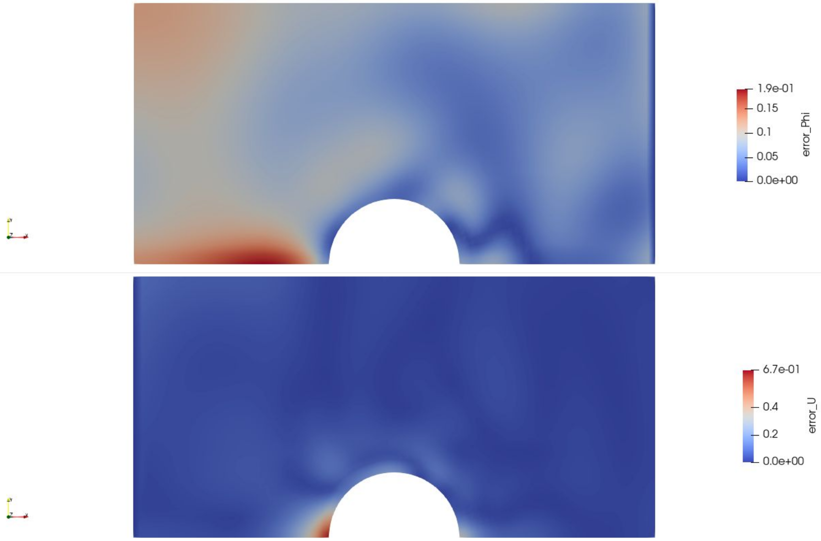
Case 2: Plain vanilla PINN



OpenFOAM®

Results: Pointwise absolute error

Case 1: Standard MLP with pure data driven loss.



Case 2: Plain vanilla PINN