



# Introduction to Physics-Informed Machine Learning with Modulus



## Agenda

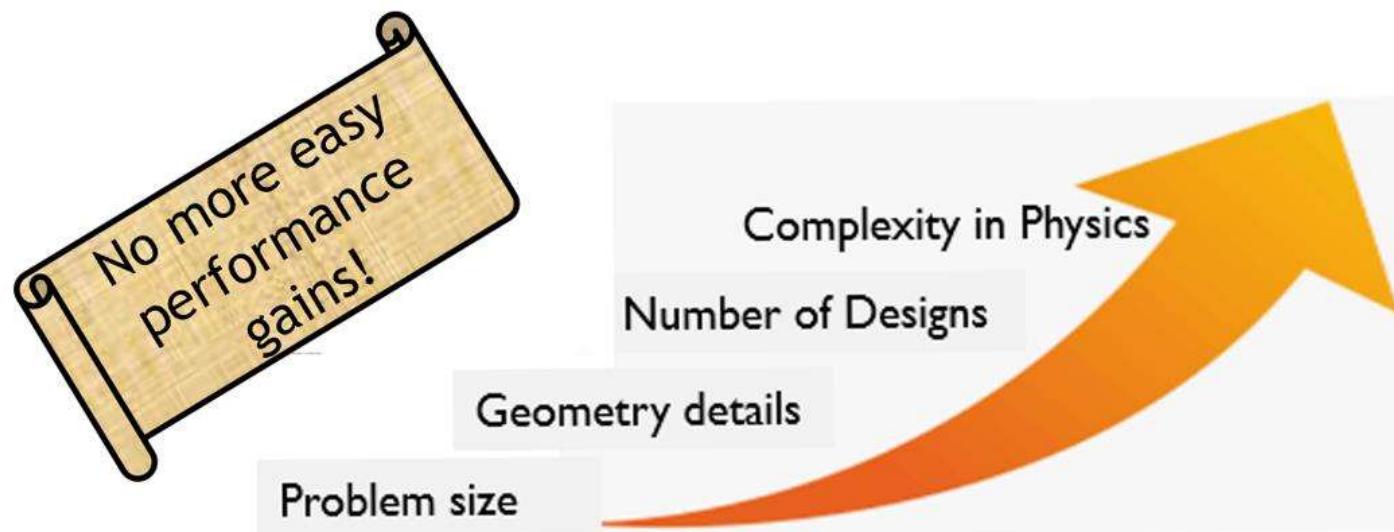
- What is Modulus?
- Modulus Architecture, Training
- Physics informed neural networks in Modulus
- Data informed neural networks in Modulus
- Modulus other features and advancements

## Saturating Performance in Traditional HPC

Simulations are getting larger and more complex

Traditional solution methods are:

- Computationally Expensive
- Plagued by Domain Discretization Techniques
- Not suitable for Data-assimilation or Inverse problems



# NVIDIA Modulus

**NVIDIA Modulus** is a neural network framework that blends the power of physics in form of governing partial differential equations (PDEs) with data to build high-fidelity, parameterized surrogate models with near-real-time latency. It offers:

- **AI Toolkit**

- Offers building blocks for developing physics-ML surrogate models

- **Scalable Performance**

- Solves larger problems faster by scaling from single GPU to multi-node implementation

- **Near-Real-Time Inference**

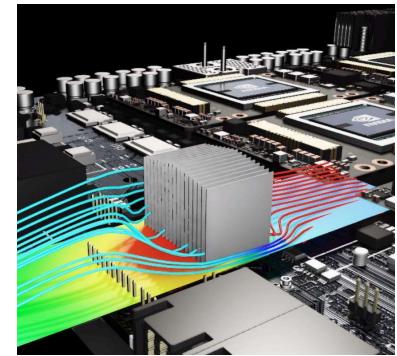
- Provides parameterized system representation that solves for multiple scenarios in near real time, trains once offline to infer in real time repeatedly

- **Easy Adoptability**

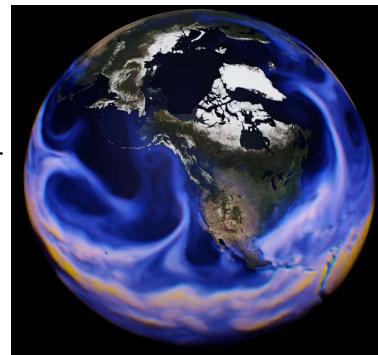
- Includes APIs for domain experts to work at a higher level of abstraction. Extensible to new applications with reference applications serving as starting points



Wind Farm Super Resolution



FPGA Heatsink Design Optimization



Extreme Weather Prediction

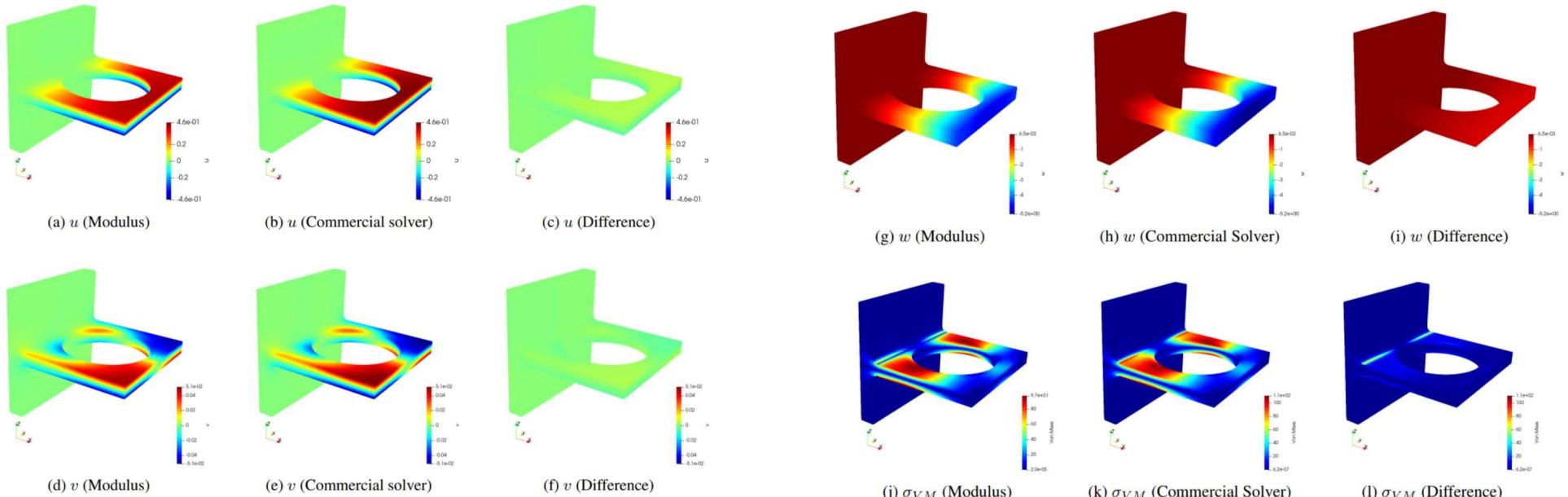


Industrial Digital Twin

# What is Modulus?

Modulus is a PDE solver

- Like traditional solvers such as Finite Element, Finite Difference, Finite Volume, and Spectral solvers, Modulus can solve PDEs

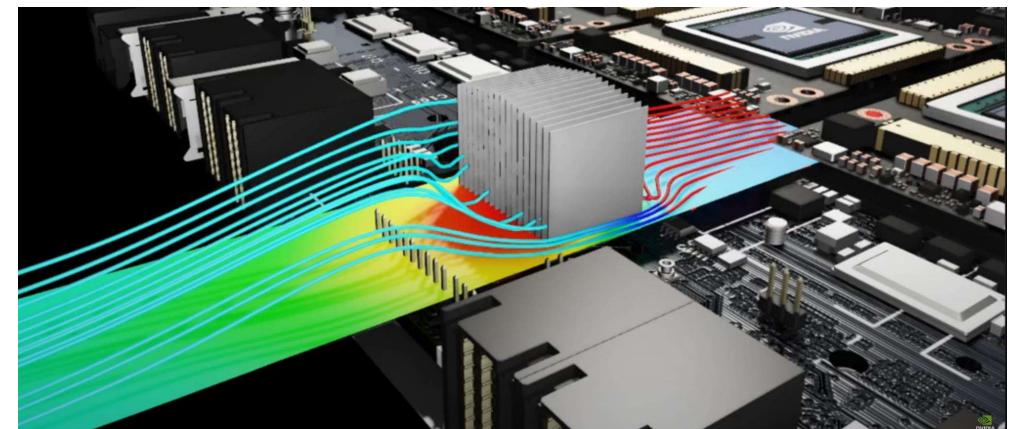
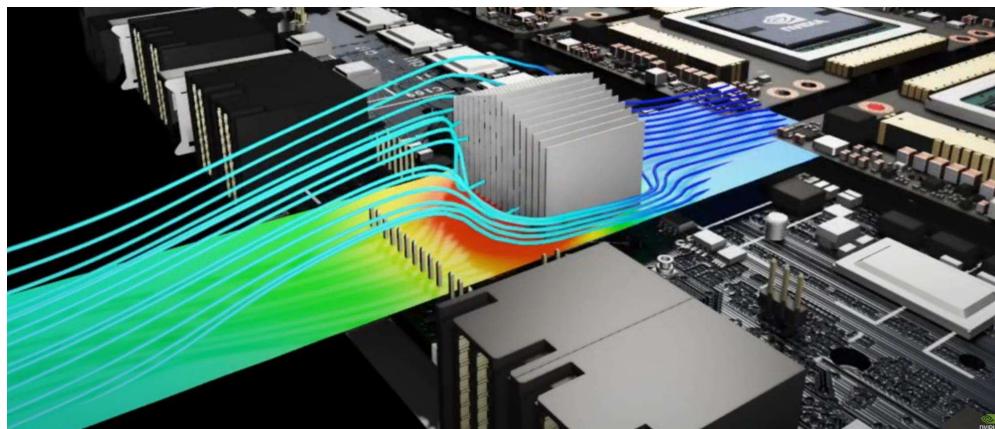


A comparison between Modulus and commercial solver results for a bracket deflection example. Linear elasticity equations are solved here.

## What is Modulus?

Modulus is a tool for efficient design optimization for multi-physics problems

- With Modulus, professionals in Manufacturing and Product Development can explore different configurations and scenarios of a model, in near-real-time by changing its parameters, allowing them to gain deeper insights about the system or product, and to perform efficient design optimization of their products.

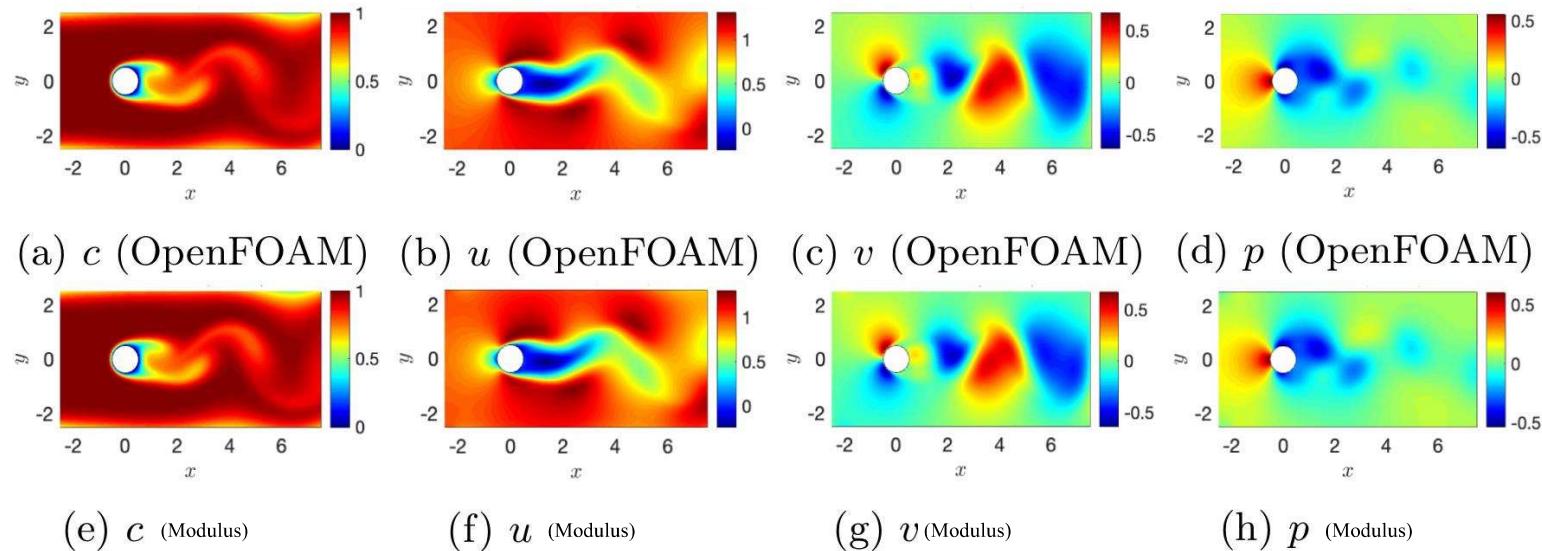


Efficient design space exploration of the heatsink of a Field-Programmable Gate Array (FPGA) using Modulus.

# What is Modulus?

Modulus is a solver for inverse problems

- Many applications in science and engineering involve inferring unknown system characteristics given measured data from sensors or imaging.
- By combining data and physics, Modulus can effectively solve inverse problems.



A comparison between Modulus and OpenFOAM results for the flow velocity, pressure and passive scalar concentration fields. Modulus has inferred the velocity and pressure fields using scattered data from passive scalar concentration.

# What is Modulus?

Modulus is a tool for developing digital twins

- A digital twin is a virtual representation (a true-to-reality simulation of physics) of a real-world physical asset or system, which is continuously updated via stream of data.
- Digital twin predicts the future state of the real-world system under varying conditions.



# What is Modulus?

Modulus is a tool for developing data-driven solutions to engineering problems

- Modulus contains a variety of APIs for developing data-driven machine learning solutions to challenging engineering systems, including:
  - Data-driven modeling of physical systems
  - Super resolution of low-fidelity results computed by traditional solvers



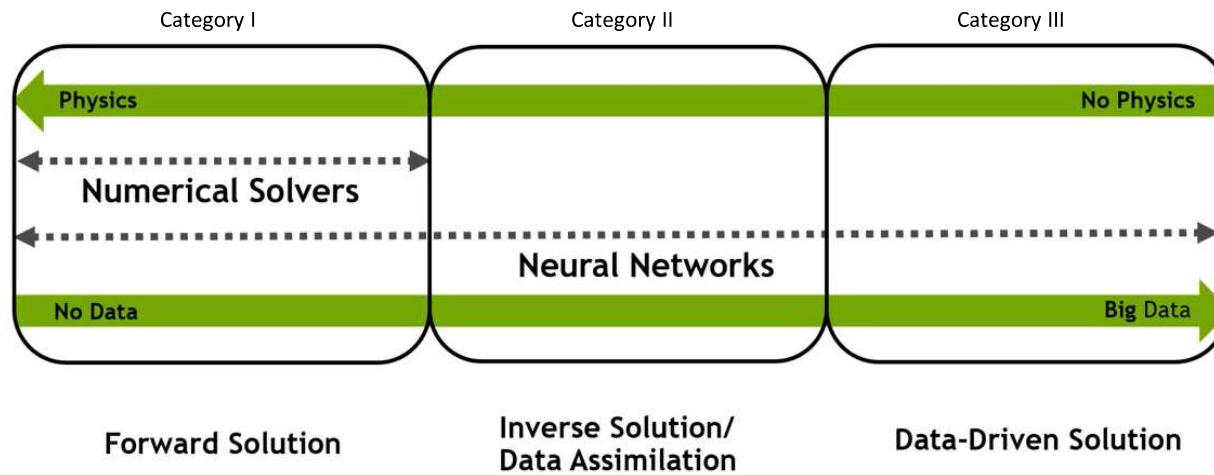
Super-resolution of flow in a wind farm using Modulus

# What is Modulus?

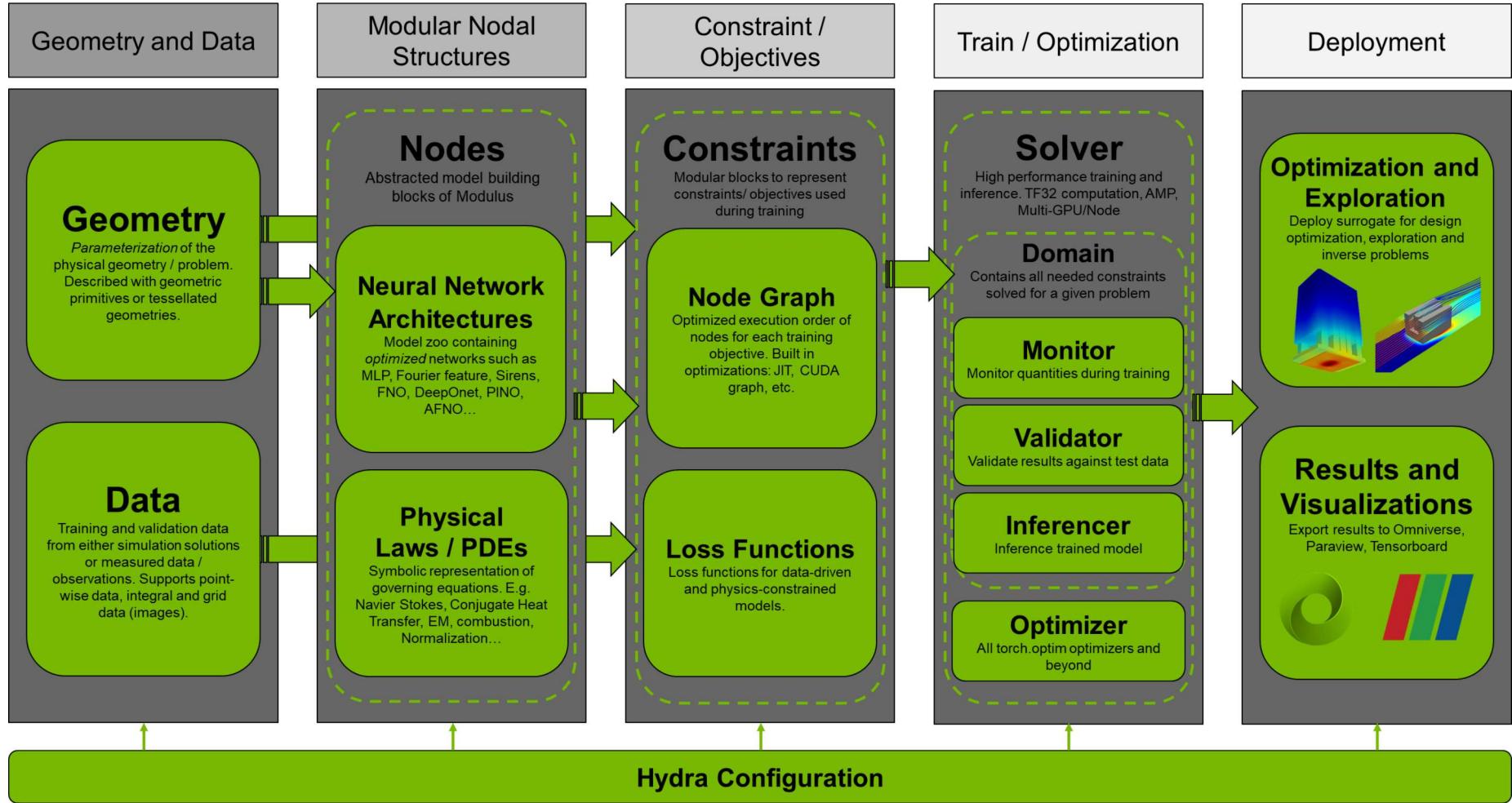
Putting it all together

- Modulus is a PDE solver (category I)
- Modulus is a tool for efficient design optimization & design space exploration (category I)
- Modulus is a solver for inverse problems (category II)
- Modulus is a tool for developing digital twins (category II)
- Modulus is a tool for developing AI solutions to engineering problems (category III)

These are all done by developing deep neural network models in Modulus that are physics-informed and/or data-informed.

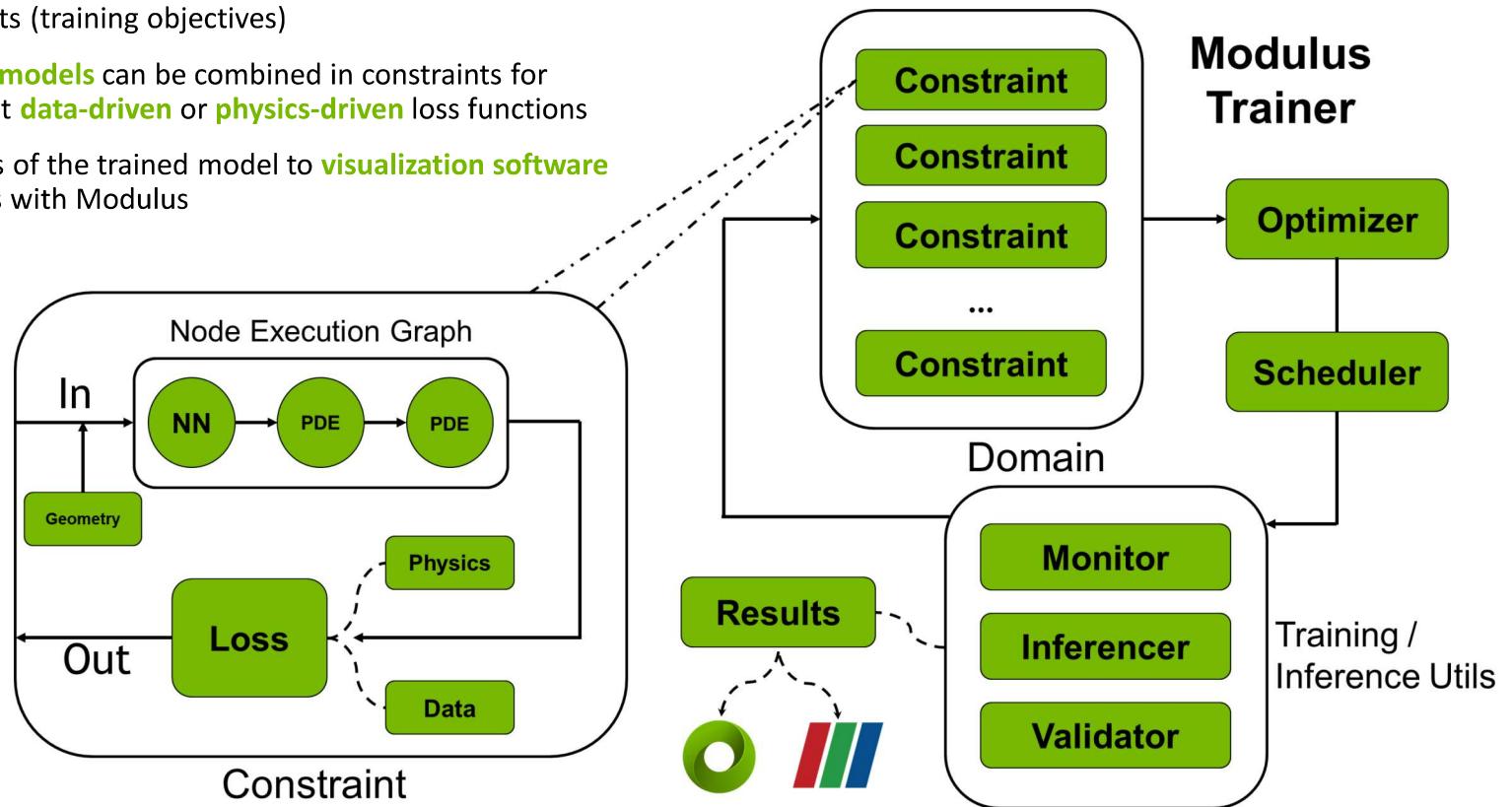


# Modulus Architecture



# Modulus Training

- Modulus allows for **complex problems** to be described through sets of constraints (training objectives)
- Multiple **nodes/models** can be combined in constraints for learning different **data-driven or physics-driven** loss functions
- Exporting results of the trained model to **visualization software** is then effortless with Modulus



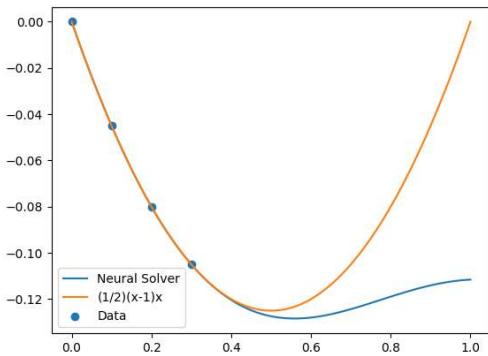
# Physics Informed Neural Networks (PINNs) in Modulus

A problem with NNs and the promise of PINNs

## Data Only

$$L_{data} = \sum_{x_i \in data} (u_{net}(x_i) - u_{true}(x_i))^2$$

$$L_{total} = L_{data}$$

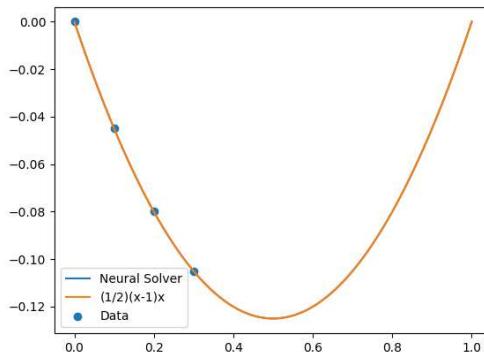


## Data + Physics

$$\frac{\delta^2 u}{\delta x^2}(x) = 1$$

$$L_{physics} = \sum_{x_j \in domain} \left( \frac{\delta^2 u_{net}}{\delta x^2}(x_j) - f(x_j) \right)^2$$

$$L_{total} = L_{data} + L_{physics}$$

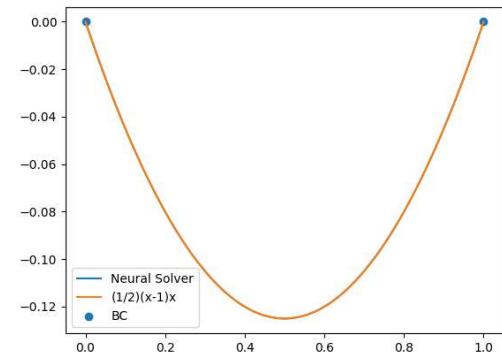


## Physics only

$$\frac{\delta^2 u}{\delta x^2}(x) = 1 \quad u(0) = u(1) = 0$$

$$L_{physics} = L_{residual} + L_{BC}$$

$$L_{total} = L_{physics}$$



# PINNs Theory: Neural Network Solver

## Problem description

- Goal: Train a neural network to satisfy the boundary conditions and differential equations by constructing an appropriate loss function
- Consider an example problem:

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

- We construct a neural network  $u_{net}(x)$  which has a single value input  $x \in \mathbb{R}$  and single value output  $u_{net}(x) \in \mathbb{R}$ .
- We assume the neural network is infinitely differentiable  $u_{net} \in C^\infty$  - Use activation functions that are infinitely differentiable

# PINNs Theory: Neural Network Solver

## Loss formulation

- Construct the loss function. We can compute the second order derivatives  $\left(\frac{\delta^2 u_{net}}{\delta x^2}(x)\right)$  using Automatic differentiation

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

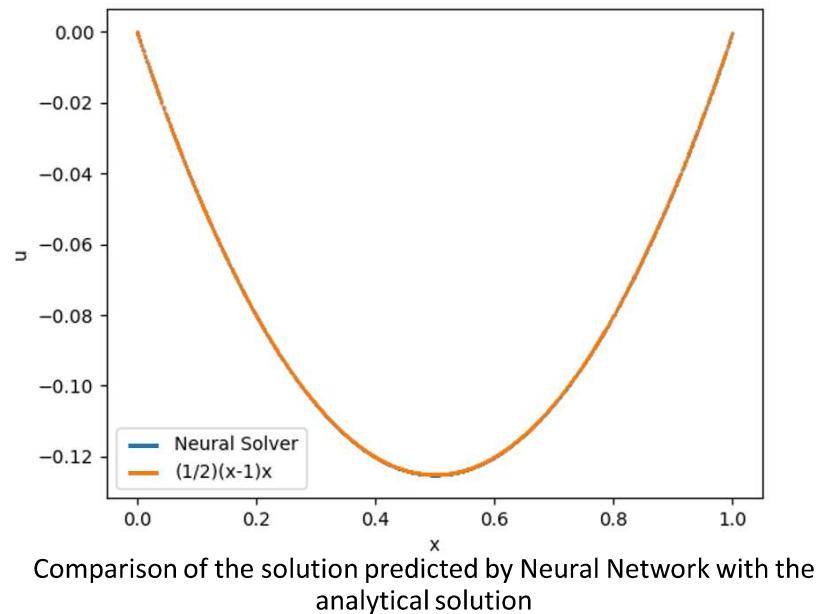
$$L_{Residual} = \int_0^1 \left( \frac{\delta^2 u_{net}}{\delta x^2}(x) - f(x) \right)^2 dx \approx \left( \int_0^1 dx \right) \frac{1}{N} \sum_{i=0}^N \left( \frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i) \right)^2$$

- Where  $x_i$  are a batch of points in the interior  $x_i \in (0, 1)$ . Total loss becomes  $L = L_{BC} + L_{Residual}$
- Minimize the loss using optimizers like Adam

# PINNs Theory: Neural Network Solver

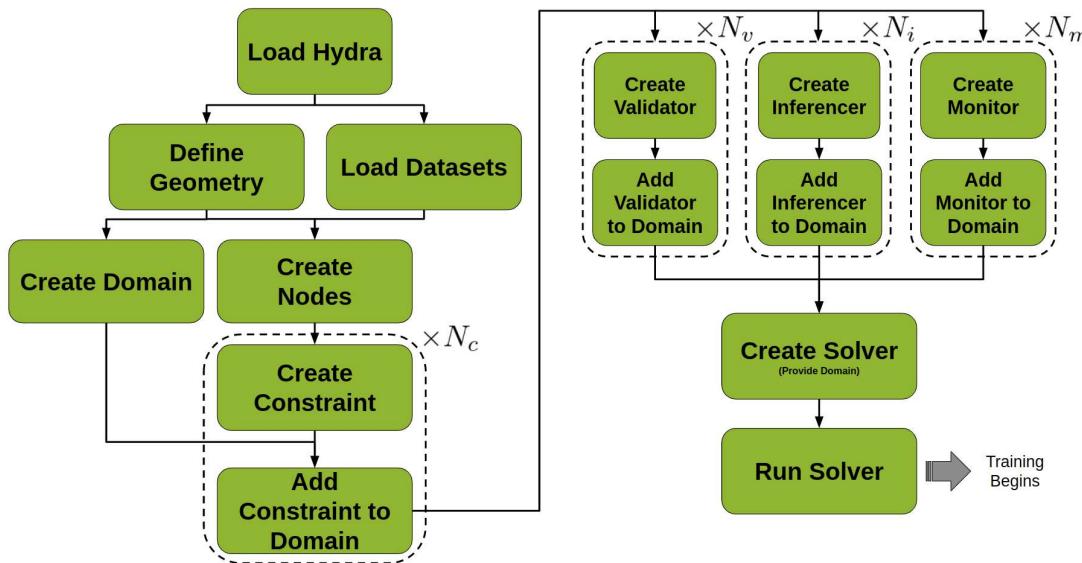
## Results

- For  $f(x) = 1$ , the true solution is  $\frac{1}{2}(x - 1)x$ . After sufficient training we have,



# Modulus: Anatomy of a project

## Overview

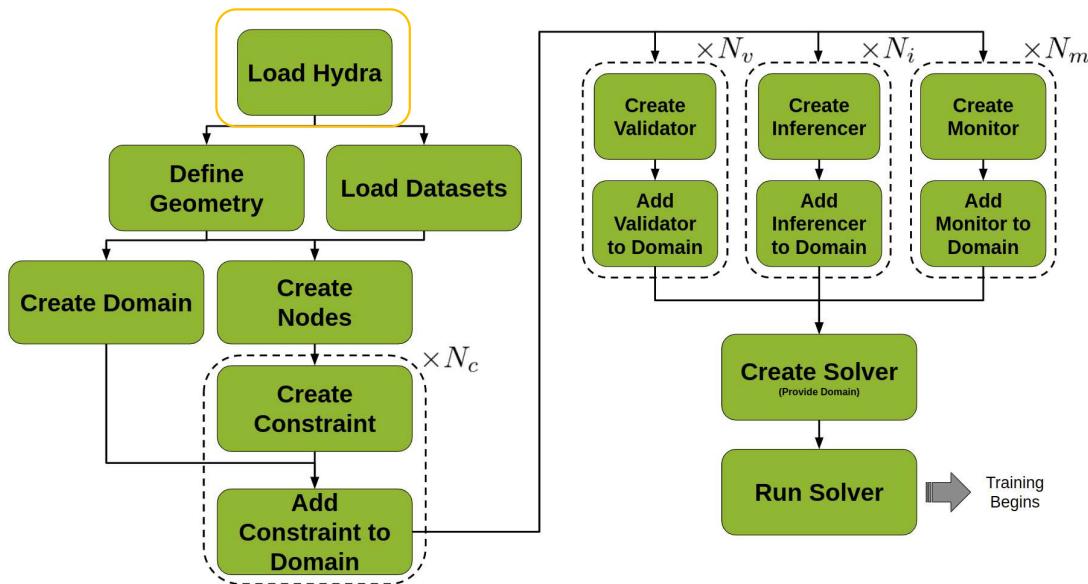


### Modulus works by:

- Writing models which include at least one adaptable function (a NN)
- Writing objective functions as a combination of these models
- Describing the geometry/dataset where the models should be evaluated
- Minimizing the objective functions by using the provided data, by sampling the geometry, or both
- Running the models to obtain the desired effect

# Modulus: Anatomy of a project

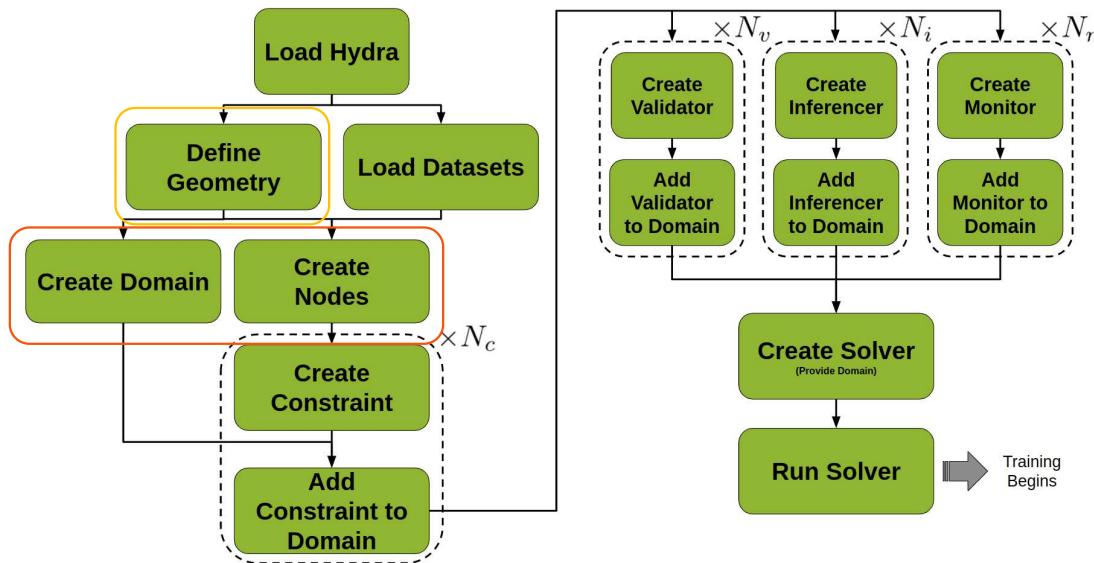
Load Hydra



```
defaults :  
- modulus_default  
- scheduler: tf_exponential_lr  
- optimizer: adam  
- loss: sum  
- _self_  
  
scheduler:  
decay_rate: 0.95  
decay_steps: 200  
  
save_filetypes : "vtk,npz"  
  
training:  
rec_results_freq : 1000  
rec_constraint_freq: 1000  
max_steps : 5000
```

# Modulus: Anatomy of a project

Create geometry, domain and nodes



```
@modulus.main(config_path="conf",
config_name="config")
def run(cfg: ModulusConfig) -> None:
```

```
# make geometry
x = Symbol("x")
geo = Line1D(0, 1)
```

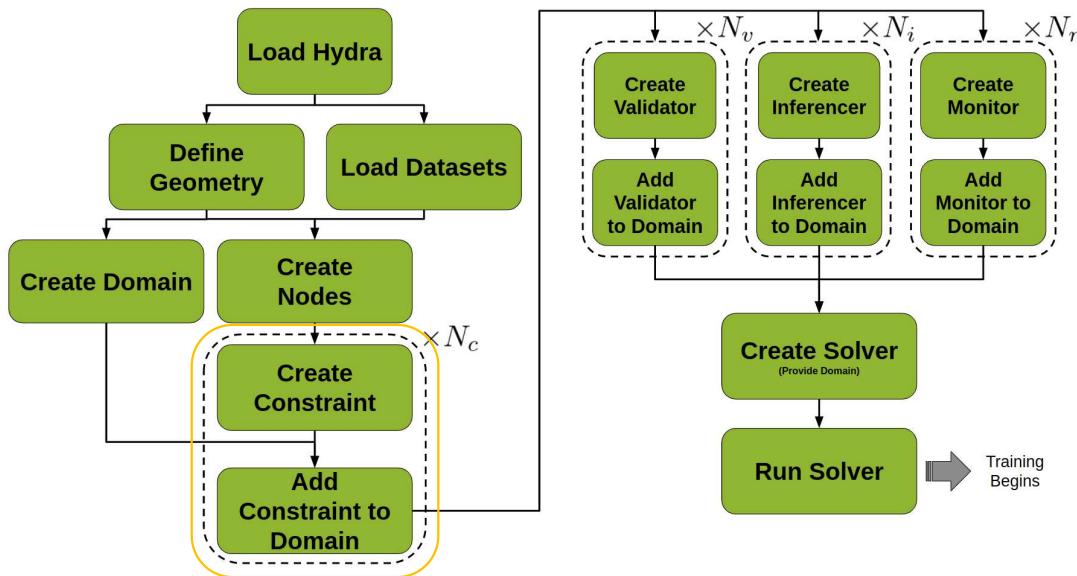
```
# make list of nodes to unroll graph on
eq = CustomPDE(f=1.0)
u_net = FullyConnectedArch(
    input_keys=[Key("x")],
    output_keys=[Key("u")],
    nr_layers=3,
    layer_size=32
)

nodes = (eq.make_nodes() +
[u_net.make_node(name="u_network")])

# make domain
domain = Domain()
```

# Modulus: Anatomy of a project

Add constraints

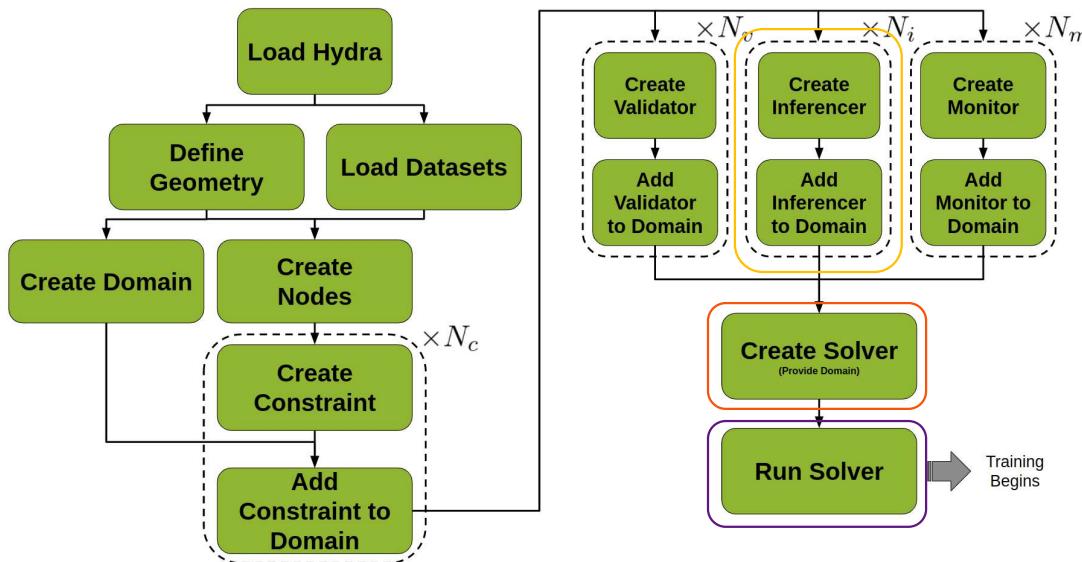


```
# add constraints to solver
# bcs
bc = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=geo,
    outvar={"u": 0},
    batch_size=2,
)
domain.add_constraint(bc, "bc")

# interior
interior = PointwiseInteriorConstraint(
    nodes=nodes,
    geometry=geo,
    outvar={"custom_pde": 0},
    batch_size=100,
    bounds={x: (0, 1)},
)
domain.add_constraint(interior, "interior")
```

# Modulus: Anatomy of a project

Create utils to visualize the results and run the solver



```
# add inferencer
x_np = np.linspace(0, 1.0, 100).reshape(-1,1)
inference = PointwiseInferencer(
    nodes=nodes,
    invar={"x": x_np},
    output_names=["u"],
)
domain.add_inferencer(inference, "inf_data")
```

```
# make solver
slv = Solver(cfg, domain)

# start solver
slv.solve()

if __name__ == "__main__":
    run()
```

```
python <script_name>.py
mpirun -np <#GPU> <script_name>.py
```

# Accessing the supplemental material

## DLI Interface and JupyterLab

The screenshot shows the DLI interface with a presentation slide. The slide features the NVIDIA logo and a complex, abstract green and black visualization of a neural network or physics-informed machine learning model. The title of the slide is "Introduction to Physics-Informed Machine Learning with Modulus" and it is presented by "Kaustubh Tangsali | Event/Date". The DLI navigation bar at the top includes links for DEEP LEARNING INSTITUTE, ONLINE COURSES, INSTRUCTOR-LED WORKSHOPS, EDUCATOR PROGRAMS, and ENTERPRISE SOLUTIONS. A user profile "KAUSTUBH" is also visible. At the bottom of the slide, there is a video control bar showing "0:00 / 1:06:54" and a "START" button.

- Navigate to the course website and begin the course
- At the bottom of the presentation, you will find a **Start** button which you can use to launch the interactive environment
- The environment may take a few minutes to get started. Once started click the **Launch** button
- You can now use this environment to run the examples provided on GPU
- Feel free to make any modifications to the examples and see how the training performs

# Solving Parameterized Problems

## Problem definition

- Consider the parameterized version of the same problem as before. Suppose we want to determine how the solution changes as we move the position on the boundary condition  $u(l) = 0$
- Parameterize the position by variable  $l \in [1, 2]$  and the problem now becomes:

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

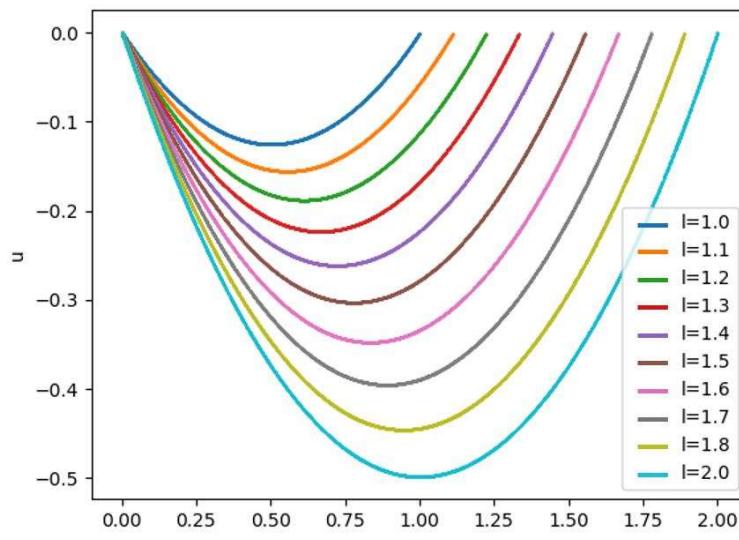
- This time, we construct a neural network  $u_{net}(x, l)$  which has  $x$  and  $l$  as input and single value output  $u_{net}(x, l) \in \mathbb{R}$ .
- The losses become

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

# Solving Parameterized Problems

## Results

- For  $f(x) = 1$ , for different values of  $l$  we have different solutions



Solution to the parametric problem

# Solving Inverse Problems

## Problem definition

- For inverse problems, we start with a set of observations and then calculate the causal factors that produced them
- For example, suppose we are given the solution  $u_{true}(x)$  at 100 random points between 0 and 1 and we want to determine the  $f(x)$  that is causing it
- Train two networks  $u_{net}(x)$  and  $f_{net}(x)$  to approximate  $u(x)$  and  $f(x)$

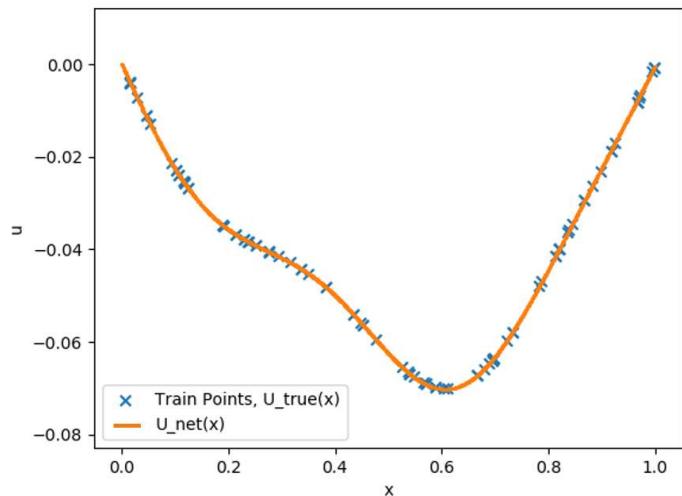
$$L_{Residual} \approx \left( \int_0^1 dx \right) \frac{1}{N} \sum_{i=0}^N \left( \frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i) \right)^2$$

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

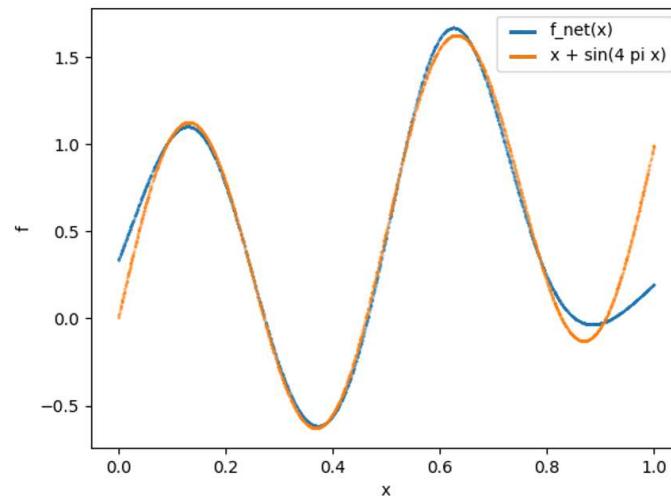
# Solving Inverse Problems

## Results

- For  $u_{true}(x) = \frac{1}{48} \left( 8x(-1 + x^2) - \frac{3 \sin(4\pi x)}{\pi^2} \right)$  the solution for  $f(x)$  is  $x + \sin(4\pi x)$



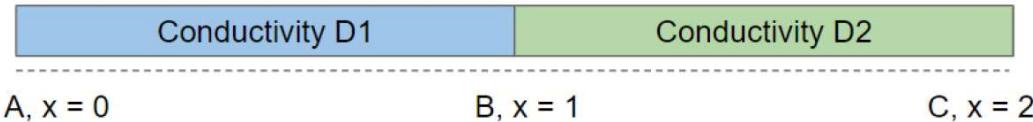
Comparison of  $u_{net}(x)$  and train points from  $u_{true}$



Comparison of the true solution for  $f(x)$  and the  $f_{net}(x)$  inverted out

## Solution to 1D diffusion

### Problem description



- Composite bar with material of conductivity  $D_1 = 10$  for  $x \in (0,1)$  and  $D_2 = 0.1$  for  $x \in (1,2)$ . Points A and C are maintained at temperatures of 0 and 100 respectively
- Equations: Diffusion equation in 1D

$$\frac{d}{dx} \left( D_1 \frac{dU_1}{dx} \right) = 0 \quad \text{When } 0 < x < 1$$

$$\frac{d}{dx} \left( D_2 \frac{dU_2}{dx} \right) = 0 \quad \text{When } 1 < x < 2$$

- Flux and field continuity at interface ( $x=1$ )

$$\begin{aligned} \left( D_1 \frac{dU_1}{dx} \right) &= \left( D_2 \frac{dU_2}{dx} \right) \\ U_1 &= U_2 \end{aligned}$$

# Solution to 1D diffusion

## Code snippets – Custom symbolic PDE

```
from sympy import Symbol, Eq, Function, Number
from modulus.eq.pde import PDE

class Diffusion(PDE):
    name = "Diffusion"

    def __init__(self, T="T", D="D", Q=0, dim=3, time=True):
        # set params
        self.T = T
        self.dim = dim
        self.time = time

        # coordinates
        x, y, z = Symbol("x"), Symbol("y"), Symbol("z")

        # time
        t = Symbol("t")

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

        # Temperature
        assert type(T) == str, "T needs to be string"
        T = Function(T)(*input_variables)

        # Diffusivity
        if type(D) is str:
            D = Function(D)(*input_variables)
        elif type(D) in [float, int]:
            D = Number(D)

        # Source
        if type(Q) is str:
            Q = Function(Q)(*input_variables)
        elif type(Q) in [float, int]:
            Q = Number(Q)

        # set equations
        self.equations = {}
        self.equations["diffusion_{} + self.T] = (
            T.diff(t) - (D * T.diff(x)).diff(x) - (D * T.diff(y)).diff(y) - (D * T.diff(z)).diff(z) - Q
        )
```

Create a child class from Modulus' PDE class

Add the `__init__()` function to define any PDE specific arguments

Symbolic input variables using sympy's `Symbol`

Dependent variables defined using sympy's `Function`

Any additional terms that potentially need to be parameterized can also be specified as dependent variables

Symbolic PDE. Derivatives are computed using sympy's functions

$$T_t = \nabla \cdot (D \nabla T) + Q$$

# Solution to 1D diffusion

## Code snippets

```

@modulus.main(config_path="conf", config_name="config")
def run(cfg: ModulusConfig) -> None:
    # make list of nodes to unroll graph on
    diff_u1 = Diffusion(T="u_1", D=D1, dim=1, time=False)
    diff_u2 = Diffusion(T="u_2", D=D2, dim=1, time=False)
    diff_in = DiffusionInterface("u_1", "u_2", D1, D2, dim=1, time=False)

    diff_net_u_1 = instantiate_arch(
        input_keys=[Key("x")],
        output_keys=[Key("u_1")],
        cfg=cfg.arch.fully_connected,
    )
    diff_net_u_2 = instantiate_arch(
        input_keys=[Key("x")],
        output_keys=[Key("u_2")],
        cfg=cfg.arch.fully_connected,
    )

    nodes = (
        diff_u1.make_nodes()
        + diff_u2.make_nodes()
        + diff_in.make_nodes()
        + [diff_net_u_1.make_node(name="u1_network", jit=cfg.jit)]
        + [diff_net_u_2.make_node(name="u2_network", jit=cfg.jit)]
    )

    # make domain add constraints to the solver
    domain = Domain()

    # sympy variables
    x = Symbol("x")

    # right hand side (x = 2) Pt c
    rhs = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=L2,
        outvar={"u_2": Tc},
        batch_size=cfg.batch_size.rhs,
        criteria=Eq(x, 2),
    )
    domain.add_constraint(rhs, "right_hand_side")

```

### Loading hydra configs

### Equation and neural network nodes

### Domain and Constraints

```

# left hand side (x = 0) Pt a
lhs = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=L1,
    outvar={"u_1": Ta},
    batch_size=cfg.batch_size.lhs,
    criteria=Eq(x, 0),
)
domain.add_constraint(lhs, "left_hand_side")

# interface 1-2
interface = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=L1,
    outvar={
        "diffusion_interface_dirichlet_u_1_u_2": 0,
        "diffusion_interface_neumann_u_1_u_2": 0,
    },
    batch_size=cfg.batch_size.interface,
    criteria=Eq(x, 1),
)
domain.add_constraint(interface, "interface")

# interior 1
interior_u1 = PointwiseInteriorConstraint(
    nodes=nodes,
    geometry=L1,
    outvar={"diffusion_u_1": 0},
    bounds={x: (0, 1)},
    batch_size=cfg.batch_size.interior_u1,
)
domain.add_constraint(interior_u1, "interior_u1")

# interior 2
interior_u2 = PointwiseInteriorConstraint(
    nodes=nodes,
    geometry=L2,
    outvar={"diffusion_u_2": 0},
    bounds={x: (1, 2)},
    batch_size=cfg.batch_size.interior_u2,
)
domain.add_constraint(interior_u2, "interior_u2")

```

Sample the boundary of geometry

Criteria for sub-sampling

Sample the interior of geometry

# Solution to 1D diffusion

## Code snippets

```
# validation data
x = np.expand_dims(np.linspace(0, 1, 100), axis=-1)
u_1 = x * Tb + (1 - x) * Ta
invar_numpy = {"x": x}
outvar_numpy = {"u_1": u_1}
val = PointwiseValidator(nodes=nodes, invar=invar_numpy, true_outvar=outvar_numpy)
domain.add_validator(val, name="val1")

# make validation data Line 2
x = np.expand_dims(np.linspace(1, 2, 100), axis=-1)
u_2 = (x - 1) * Tc + (2 - x) * Tb
invar_numpy = {"x": x}
outvar_numpy = {"u_2": u_2}
val = PointwiseValidator(nodes=nodes, invar=invar_numpy, true_outvar=outvar_numpy)
domain.add_validator(val, name="val2")
```

```
# make monitors
invar_numpy = {"x": [[1.0]]}
monitor = PointwiseMonitor(
    invar_numpy,
    output_names=["u_1_x"],
    metrics={"flux_u1": lambda var: torch.mean(var["u_1_x"])},
    nodes=nodes,
    requires_grad=True,
)
domain.add_monitor(monitor)

monitor = PointwiseMonitor(
    invar_numpy,
    output_names=["u_2_x"],
    metrics={"flux_u2": lambda var: torch.mean(var["u_2_x"])},
    nodes=nodes,
    requires_grad=True,
)
domain.add_monitor(monitor)
```

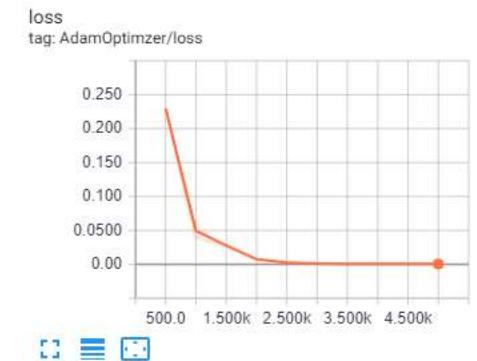
```
# make solver
slv = Solver(cfg, domain)

# start solver
slv.solve()
```

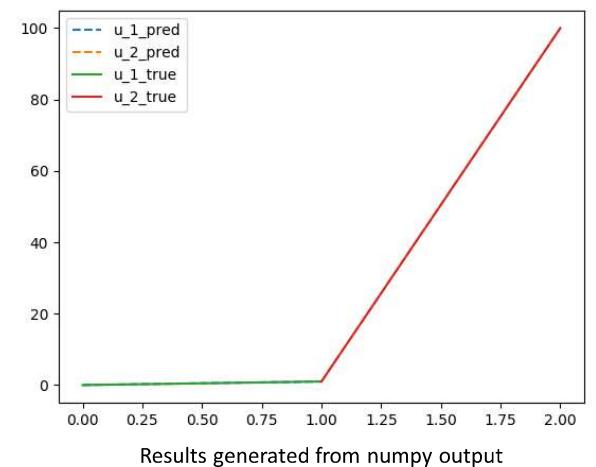
Validators to compare with experimental/analytical /solver data

Monitor the quantities of interest during the runtime

Solver



Tensorboard visualization of loss curves



# Parameterized Solution to 1D diffusion

## Problem description and code snippets

- Composite bar with material of conductivity  $D_1$  for  $x \in (0,1)$  and  $D_2 = 0.1$  for  $x \in (1,2)$ .
- Solve the problem for multiple values of  $D_1$  in the range (5, 25) in a single training
- Same boundary and interface conditions as before

```
# params for domain
L1 = Line1D(0, 1)
L2 = Line1D(1, 2)

D1 = Symbol("D1")
D1_range = {D1: (5, 25)}

@modulus.main(config_path="conf", config_name="config_param")
def run(cfg: ModulusConfig) -> None:
    # make list of nodes to unroll graph on
    diff_u1 = Diffusion(T="u_1", D=D1, dim=1, time=False)
    diff_u2 = Diffusion(T="u_2", D=D2, dim=1, time=False)
    diff_in = DiffusionInterface("u_1", "u_2", "D1", D2, dim=1, time=False)

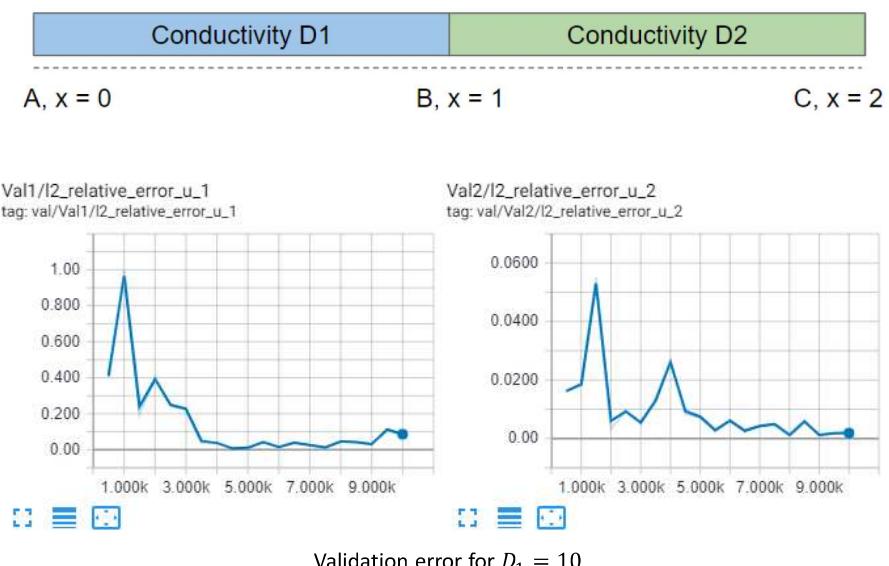
    diff_net_u_1 = instantiate_arch(
        input_keys=[Key("x"), Key("D1")],
        output_keys=[Key("u_1")],
        cfg=cfg.arch.fully_connected,
    )
    diff_net_u_2 = instantiate_arch(
        input_keys=[Key("x"), Key("D1")],
        output_keys=[Key("u_2")],
        cfg=cfg.arch.fully_connected,
    )

    # right hand side (x = 2) Pt c
    rhs = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=L2,
        outvar={"u_2": Tc},
        batch_size=cfg.batch_size.rhs,
        criteria=Eq(x, 2),
        parameterization=Parameterization(D1_range),
    )
    domain.add_constraint(rhs, "right_hand_side")
```

Symbolically parameterize the variables of choice (geometric/physical) and setup the architecture

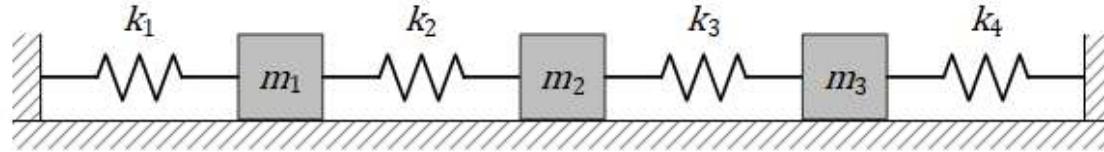
Specify the appropriate parameterization to the constraints

Fairly simple changes to go from single forward simulation to parameterized simulation



## Solution to ODEs – Coupled Spring Mass System

Problem description



- Three masses connected by four springs
- System's equations (ordinary differential equations):

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

- For given values masses, spring constants and boundary conditions

$$\begin{aligned}[m_1, m_2, m_3] &= [1, 1, 1] \\[k_1, k_2, k_3, k_4] &= [2, 1, 1, 2] \\[x_1(0), x_2(0), x_3(0)] &= [1, 0, 0] \\[x'_1(0), x'_2(0), x'_3(0)] &= [0, 0, 0]\end{aligned}$$

# Solution to ODEs – Coupled Spring Mass System

## Code snippets

```

@modulus.main(config_path="conf", config_name="config")
def run(cfg: ModulusConfig) -> None:
    # make list of nodes to unroll graph on
    sm = SpringMass(k=[2, 1, 1, 2], m=[1, 1, 1])
    sm_net = instantiate_arch(
        input_keys=[Key("t")],
        output_keys=[Key("x1"), Key("x2"), Key("x3")],
        cfg=cfg.arch.fully_connected,
    )
    nodes = sm.make_nodes() + [
        sm_net.make_node(name="spring_mass_network", jit=cfg.jit)
    ]
    # add constraints to solver
    # make geometry
    geo = Point1D(0)
    t_max = 10.0
    t_symbol = Symbol("t")
    x = Symbol("x")
    time_range = {t_symbol: (0, t_max)}

    # make domain
    domain = Domain()

    # initial conditions
    IC = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=geo,
        outvar={"x1": 1.0, "x2": 0, "x3": 0, "x1_t": 0, "x2_t": 0, "x3_t": 0},
        batch_size=cfg.batch_size.IC,
        lambda_weighting={
            "x1": 1.0,
            "x2": 1.0,
            "x3": 1.0,
            "x1_t": 1.0,
            "x2_t": 1.0,
            "x3_t": 1.0,
        },
        parameterization=Parameterization({t_symbol: 0}),
    )
    domain.add_constraint(IC, name="IC")

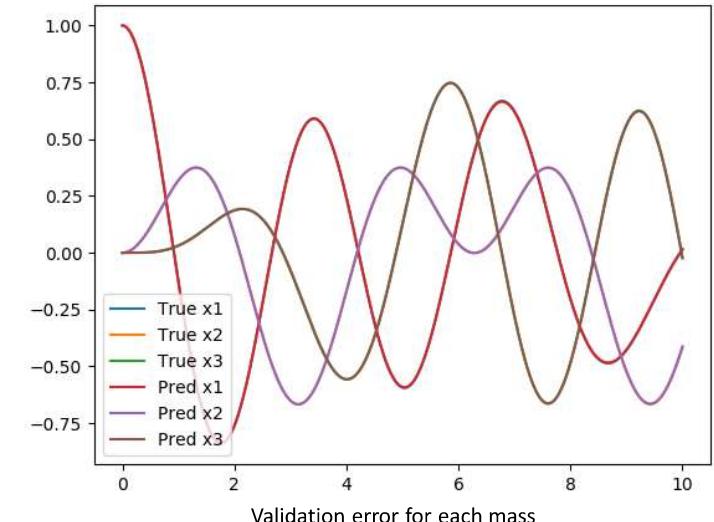
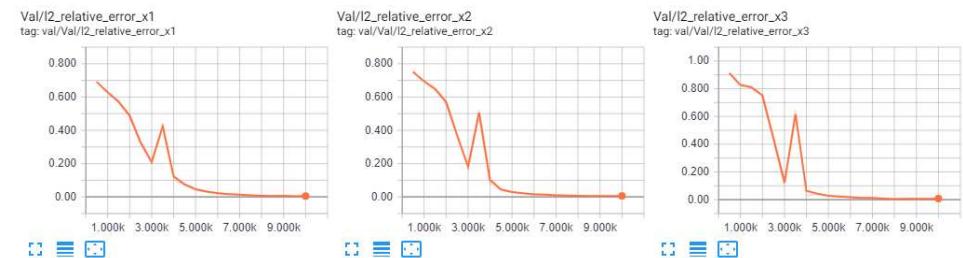
    # solve over given time period
    interior = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=geo,
        outvar={"ode_x1": 0.0, "ode_x2": 0.0, "ode_x3": 0.0},
        batch_size=cfg.batch_size.interior,
        parameterization=Parameterization(time_range),
    )
    domain.add_constraint(interior, "interior")

```

Nodes for a simple transient problem using **continuous time approach**

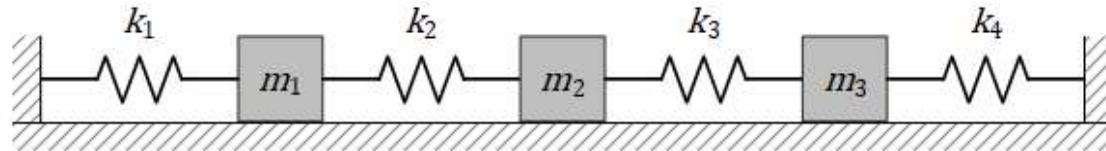
Under **continuous time approach**, time can be modeled as any other parametrized variable

Simple transient problem using **continuous time approach**. Advanced approaches like **moving-time-window** etc. also available for more complex problems.



## Inverse Problem – Coupled Spring Mass System

Problem description



- For the same system, assume we know the analytical solution which is given by:

$$x_1(t) = \frac{1}{6}\cos(t) + \frac{1}{2}\cos(\sqrt{3}t) + \frac{1}{3}\cos(2t);$$
$$x_2(t) = \frac{2}{6}\cos(t) - \frac{1}{3}\cos(2t);$$
$$x_3(t) = \frac{1}{6}\cos(t) - \frac{1}{2}\cos(\sqrt{3}t) + \frac{1}{3}\cos(2t)$$

- With the above data and the values for  $m_2, m_3, k_1, k_2, k_3$  same as before, use the neural network to find the values of  $m_1$  and  $k_4$

# Inverse Problem – Coupled Spring Mass System

Code snippets

```
@modulus.main(config_path="conf", config_name="config_inverse")
def run(cfg: ModulusConfig) -> None:
    # prepare data
    t_max = 10.0
    deltaT = 0.01
    t = np.arange(0, t_max, deltaT)
    t = np.expand_dims(t, axis=-1)

    invar_numpy = {"t": t}
    outvar_numpy = {
        "x1": (1 / 6) * np.cos(t)
        + (1 / 2) * np.cos(np.sqrt(3) * t)
        + (1 / 3) * np.cos(2 * t),
        "x2": (2 / 6) * np.cos(t)
        + (0 / 2) * np.cos(np.sqrt(3) * t)
        - (1 / 3) * np.cos(2 * t),
        "x3": (1 / 6) * np.cos(t)
        - (1 / 2) * np.cos(np.sqrt(3) * t)
        + (1 / 3) * np.cos(2 * t),
    }
    outvar_numpy.update({"ode_x1": np.full_like(invar_numpy["t"], 0)})
    outvar_numpy.update({"ode_x2": np.full_like(invar_numpy["t"], 0)})
    outvar_numpy.update({"ode_x3": np.full_like(invar_numpy["t"], 0)})
```

Analytical data

```
# make list of nodes to unroll graph on
sm = SpringMass(k=(2, 1, 1, "k4"), m=("m1", 1, 1))
sm_net = instantiate_arch(
    input_keys=[Key("t")],
    output_keys=[Key("x1"), Key("x2"), Key("x3")],
    cfg=cfg.arch.fully_connected,
)
invert_net = instantiate_arch(
    input_keys=[Key("t")],
    output_keys=[Key("m1"), Key("k4")],
    cfg=cfg.arch.fully_connected,
)
```

Additional network  
to invert out the  
unknowns

```
# data and pdes
data = PointwiseConstraint.from_numpy(
    nodes=nodes,
    invar=invar_numpy,
    outvar=outvar_numpy,
    batch_size=cfg.batch_size.data,
)
domain.add_constraint(data, name="Data")
```

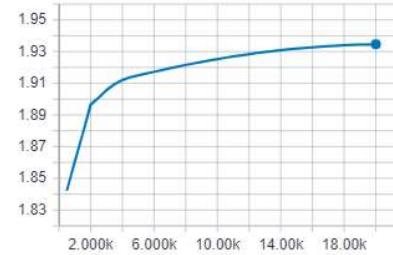
```
# add monitors
monitor = PointwiseMonitor(
    invar_numpy,
    output_names=["m1"],
    metrics={"mean_m1": lambda var: torch.mean(var["m1"])},
    nodes=nodes,
)
domain.add_monitor(monitor)

monitor = PointwiseMonitor(
    invar_numpy,
    output_names=["k4"],
    metrics={"mean_k4": lambda var: torch.mean(var["k4"])},
    nodes=nodes,
)
domain.add_monitor(monitor)
```

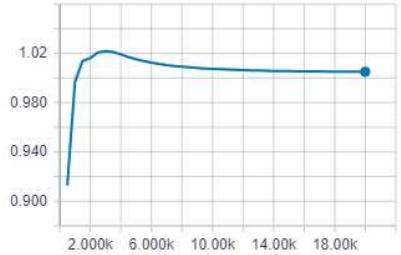
Assimilate the data  
using  
PointwiseConstraint

Monitors to infer the  
inverted quantities

GlobalMonitor/average\_k4  
tag: monitor/GlobalMonitor/average\_k4



GlobalMonitor/average\_m1  
tag: monitor/GlobalMonitor/average\_m1



Results

## Data-driven analysis – Surrogate Model for Darcy Flow

### Problem description

- Use Fourier Neural Operator (FNO) and its variants to develop a surrogate model that learns a mapping between a permeability field and pressure field,  $\mathbf{K} \rightarrow \mathbf{U}$ , for a distribution of permeability fields  $\mathbf{K} \sim p(\mathbf{K})$ .
- The Darcy PDE is a second order, elliptical PDE with the following form:

$$-\nabla \cdot (k(\mathbf{x}) \nabla u(\mathbf{x})) = f(\mathbf{x}), \quad \mathbf{x} \in D,$$

- Where  $u(\mathbf{x})$  is the flow pressure,  $k(\mathbf{x})$  is the permeability field and  $f(\cdot)$  is the forcing function.
- The Darcy flow can parameterize a variety of systems including flow through porous media, elastic materials and heat conduction.
- Here, we will define the domain as a 2D unit square  $D = \{x, y \in (0,1)\}$  with the boundary condition  $u(\mathbf{x}) = 0, \mathbf{x} \in \partial D$ .
- Both the permeability and flow fields are discretized into a 2D matrix  $\mathbf{K}, \mathbf{u} \in \mathbb{R}^{N \times N}$ .

# Darcy Flow – Fourier Neural Operators (FNO)

## Theory

- FNO can be used to parameterize solutions for a distribution of PDE solutions. The key feature of FNO is the spectral convolutions: operations that place the integral kernel in Fourier space:

$$(K(\mathbf{w})\phi)(x) = F^{-1}(R_{\mathbf{w}} \cdot (F)\phi)(x), \quad \forall x \in D$$

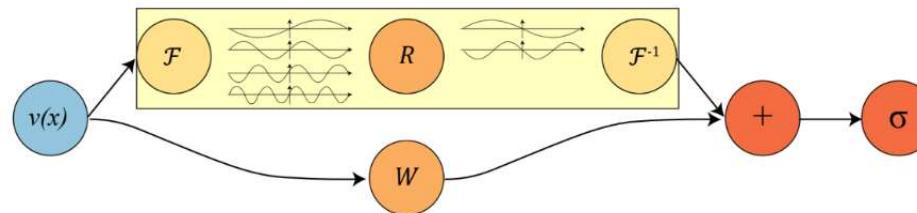
- $R_{\mathbf{w}}$  is the transformation containing the learnable parameters  $\mathbf{w}$ . This operator is calculated over the entire *structured Euclidean* domain  $D$ . Fast Fourier Transform (FFT) is used to perform the Fourier transforms efficiently and the resulting transformation  $R_{\mathbf{w}}$  is a finite size matrix of learnable weights.
- In spectral convolution, the Fourier coefficients are truncated to only the lower modes which allows explicit control over the dimensionality of the spectral space and linear operator.

- The FNO model is a composition of a fully-connected “lifting” layer,  $L$  spectral convolutions with point-wise linear skip connections and a decoding point-wise fully-connected neural network at the end:

$$u_{\text{net}}(\Phi; \theta) = Q \circ \sigma(\mathbf{W}_L + \mathbf{K}_L) \circ \dots \circ \sigma(\mathbf{W}_1 + \mathbf{K}_1) \circ P(\Phi), \quad \Phi = \{\phi(x); \forall x \in D\}$$

- Here,  $\sigma(\mathbf{W}_i + \mathbf{K}_i)$  is the spectral convolution layer  $i$  with the point-wise linear transform  $\mathbf{W}_i$  and activation function  $\sigma(\cdot)$ .  $P$  is the point-wise lifting network that projects the input into a higher dimensional latent space.  $Q$  is the point-wise fully-connected decoding network.

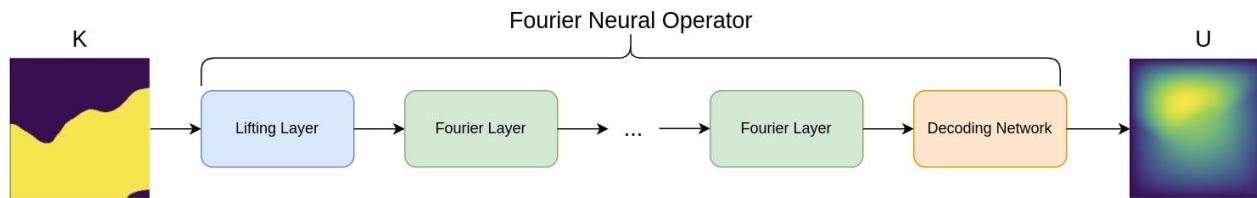
- Since all fully-connected components of FNO are point-wise operations, the model is invariant to the dimensionality of the input.



Source: Li et al., [2010.08895.pdf \(arxiv.org\)](https://arxiv.org/pdf/2010.08895.pdf)

# Darcy Flow – Fourier Neural Operators (FNO)

## Code Snippets



```
@modulus.main(config_path="conf", config_name="config_FNO")
def run(cfg: ModulusConfig) -> None:

    # load training/ test data
    input_keys = [Key("coeff", scale=(7.48360e00, 4.49996e00))]
    output_keys = [Key("sol", scale=(5.74634e-03, 3.88433e-03))]

    download_FNO_dataset("Darcy_241", outdir="datasets/")
    train_path = to_absolute_path(
        "datasets/Darcy_241/piececonst_r241_N1024_smooth1.hdf5"
    )
    test_path = to_absolute_path(
        "datasets/Darcy_241/piececonst_r241_N1024_smooth2.hdf5"
    )

    # make datasets
    train_dataset = HDF5GridDataset(
        train_path, invar_keys=["coeff"], outvar_keys=["sol"], n_examples=1000
    )
    test_dataset = HDF5GridDataset(
        test_path, invar_keys=["coeff"], outvar_keys=["sol"], n_examples=100
    )

    # make list of nodes to unroll graph on
    model = instantiate_arch(
        input_keys=input_keys,
        output_keys=output_keys,
        cfg=cfg.arch.fno,
    )
    nodes = model.make_nodes(name="FNO", jit=cfg.jit)
```

Lazy data loading (in addition to eager loading) available for large datasets

Instantiate the FNO model. Network parameters handled in configs.

```
defaults :
    - modulus_default
    - arch:
        - fno
    - scheduler: tf_exponential_lr
    - optimizer: adam
    - loss: sum
    - _self_
    jit: false

arch:
    fno:
        dimension: 2
        nr_fno_layers: 4
        fno_layer_size: 32
        fno_modes: 12
        padding: 9
        output_fc_layer_sizes:
            - 128

scheduler:
    decay_rate: 0.95
    decay_steps: 1000

training:
    rec_results_freq : 1000
    max_steps : 10000

batch_size:
    grid: 32
    validation: 32

Hydra configs for the Darcy FNO problem
```

# Darcy Flow – Fourier Neural Operators (FNO)

## Code Snippets

```
# make domain
domain = Domain()

# add constraints to domain
supervised = SupervisedGridConstraint(
    nodes=nodes,
    dataset=train_dataset,
    batch_size=cfg.batch_size.grid,
    num_workers=4, # number of parallel data loaders
)
domain.add_constraint(supervised, "supervised")
```

Supervised grid constraints

```
# add validator
val = GridValidator(
    nodes,
    dataset=test_dataset,
    batch_size=cfg.batch_size.validation,
    plotter=GridValidatorPlotter(n_examples=5),
)
domain.add_validator(val, "test")
```

Grid Validators

```
# make solver
slv = Solver(cfg, domain)

# start solver
slv.solve()
```

- Note

- The constraint will construct the data loader automatically.
  - Modulus constraints can instantly be scaled to multi-node/multi-GPU training.
  - Optimizations such as torch script/CUDA graphs will be performed under-the-hood.
  - Batch size is controlled in the Hydra configuration file.
- These are the two core components of setting up basic data-driven problem in Modulus.

# Darcy Flow – Fourier Neural Operators (FNO)

## Results and discussion

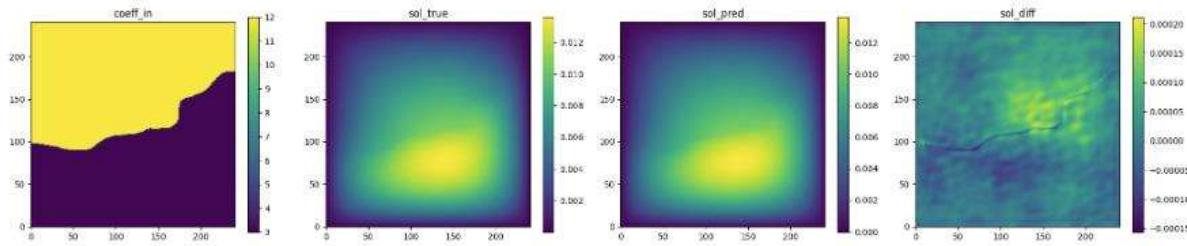


Fig. 61 FNO validation prediction 1. (Left to right) Input permeability, true pressure, predicted pressure, error.

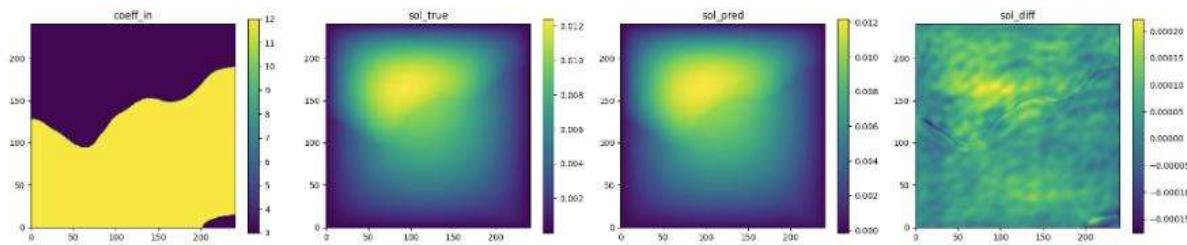


Fig. 62 FNO validation prediction 2. (Left to right) Input permeability, true pressure, predicted pressure, error.

FNO accurately learns the solution of this system.

Modulus supports the visualization of results through images (matplotlib), Tensorboard, VTK files and Omniverse for select problems.

For more information, please refer to the official [Modulus user guide example](#).

# Darcy Flow – Adaptive Fourier Neural Operators (AFNO)

## Theory

- AFNO combines the Fourier Neural Operator (FNO) with the powerful Vision Transformer (ViT) model for image processing. The ViT and related variants of transformer models have achieved SOTA performance in image processing tasks. The multi-head self-attention (MHSA) mechanism of the ViT is key to its impressive performance.
- The first step in the architecture involves dividing the input image into a regular grid with  $h \times w$  equal sized patches of size  $p \times p$ . The parameter  $p$  is referred to as the patch size. For simplicity, we consider a single channel image. Each patch is embedded into a token of size  $d$ , the embedding dimension. The patch embedding operation results in a token tensor ( $X_{h \times w \times d}$ ) of size  $h \times w \times d$ . The patch size and embedding dimension are user selected parameters. A smaller patch size allows the model to capture fine scale details better while increasing the computational cost of training the model. A higher embedding dimension also increases the parameter count of the model. The token tensor is then processed by multiple layers of the transformer architecture performing spatial and channel mixing. The AFNO architecture implements the following operations in each layer.
  - The token tensor is first transformed to the Fourier domain with
$$Z_{m,n} = [\text{DFT}(X)]_{m,n}$$
  - Here,  $m, n$  is the index the patch location and DFT denotes a 2D discrete Fourier transform.

# Darcy Flow – Adaptive Fourier Neural Operators (AFNO)

## Code Snippets

```

@modulus.main(config_path="conf", config_name="config_AFNO")
def run(cfg: ModulusConfig) -> None:

    # Load training/ test data
    input_keys = [Key("coeff", scale=(7.48360e00, 4.49996e00)),
                  output_keys = [Key("sol", scale=(5.74634e-03, 3.88433e-03))]

    download_FNO_dataset("Darcy_241", outdir="datasets/")
    invar_train, outvar_train = load_FNO_dataset(
        "datasets/Darcy_241/piececonst_r241_N1024_smooth1.hdf5",
        [k.name for k in input_keys],
        [k.name for k in output_keys],
        n_examples=1000,
    )
    invar_test, outvar_test = load_FNO_dataset(
        "datasets/Darcy_241/piececonst_r241_N1024_smooth2.hdf5",
        [k.name for k in input_keys],
        [k.name for k in output_keys],
        n_examples=100,
    )

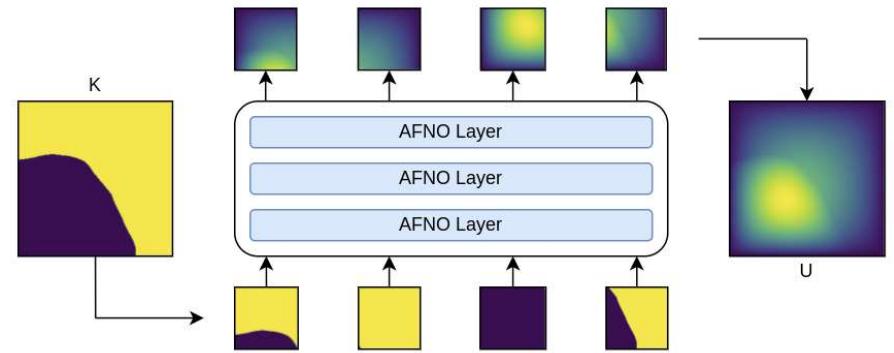
    # get training image shape
    img_shape = [
        next(iter(invar_train.values())).shape[-2],
        next(iter(invar_train.values())).shape[-1],
    ]

    # crop out some pixels so that img_shape is divisible by patch_size of AFNO
    img_shape = [s - s % cfg.arch.afno.patch_size for s in img_shape]
    print(f"cropped img_shape: {img_shape}")
    for d in (invar_train, outvar_train, invar_test, outvar_test):
        for k in d:
            d[k] = d[k][:, :, : img_shape[0], : img_shape[1]]
    print(f"\{k\}: {d[k].shape}")

```

Lazy loading like FNO example

Crop the data to make divisible by patch size



```

# make domain
domain = Domain()

# add constraints to domain
supervised = SupervisedGridConstraint(
    nodes=nodes,
    dataset=train_dataset,
    batch_size=cfg.batch_size.grid,
    num_workers=4, # number of parallel data loaders
)
domain.add_constraint(supervised, "supervised")

# add validator
val = GridValidator(
    nodes,
    dataset=test_dataset,
    batch_size=cfg.batch_size.validation,
    plotter=GridValidatorPlotter(n_examples=5),
)
domain.add_validator(val, "test")

```

Constraints and Validators same as FNO example.  
Without any changes to the core constraint/validator, experiment with different architectures with ease

# Darcy Flow – Adaptive Fourier Neural Operators (AFNO)

## Results and discussion

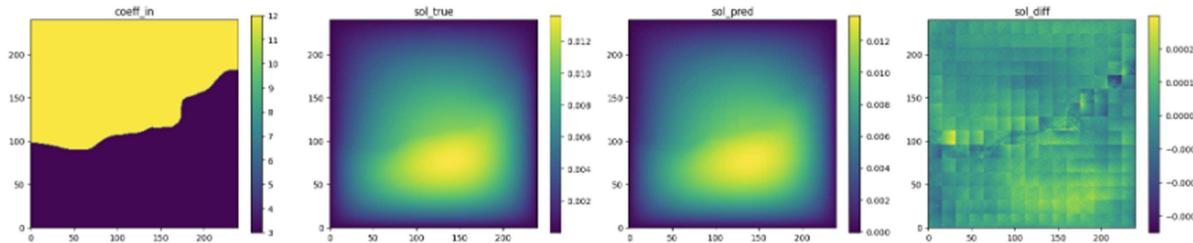


Fig. 65 AFNO validation prediction 1. (Left to right) Input permeability, true pressure, predicted pressure, error.

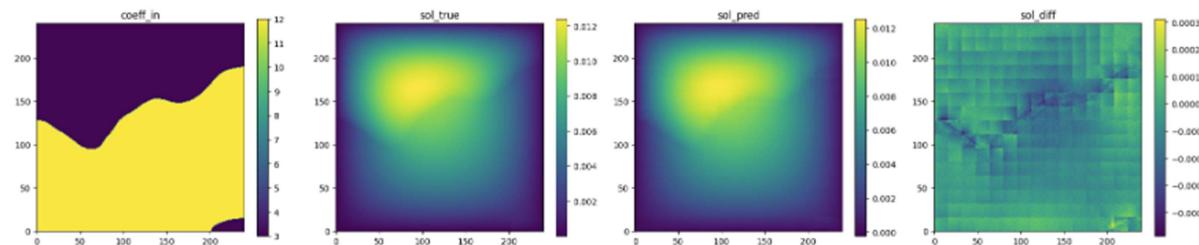
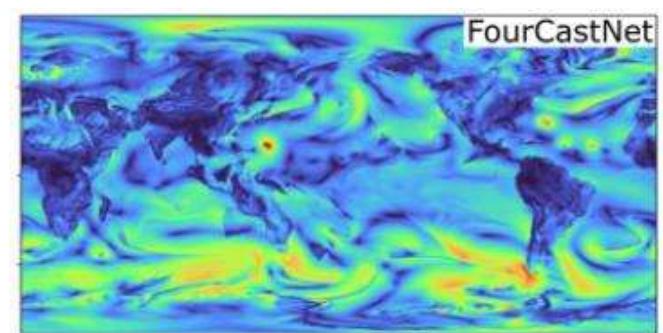


Fig. 66 AFNO validation prediction 2. (Left to right) Input permeability, true pressure, predicted pressure, error.

AFNO accurately learns the solution of this system.

For more information, please refer to the official [Modulus user guide example](#).

AFNO is the key backboard for NVIDIA's groundbreaking [FourCastNet](#):



# Darcy Flow – Physics Informed Neural Operators (PINO)

## Theory

- PINO incorporates a PDE loss to the Fourier Neural Operator. This improves interpretability and reduces the amount of data required to train a surrogate model, since the PDE loss constrains the solution space.
- In general, the PDE loss involves computing the PDE operator which in turn involves computing the partial derivatives of the Fourier Neural Operator ansatz, which is nontrivial. The key set of innovations in the PINO are the various ways to compute the partial derivatives of the operator ansatz. The PINO framework implements the differentiation in three different ways.
  - Numerical differentiation using a finite difference Method (FDM).
  - Numerical differentiation computed via spectral derivative.
  - Hybrid differentiation based on a combination of first-order “exact” and second-order FDM derivatives.

# Darcy Flow – Physics Informed Neural Operators (PINO)

## Code snippets

- Setting up a PINO problem in Modulus requires setting up a custom PDE loss that can compute on grid data. Readers are referred to the supplemental Jupyter notebooks for the code snippets for the Loss definition.

```
@modulus.main(config_path="conf", config_name="config_PINO")
def run(cfg: ModulusConfig) -> None:

    # Load training/ test data
    input_keys = [
        Key("coeff", scale=(7.48360e00, 4.49996e00)),
        Key("Kcoeff_x"),
        Key("Kcoeff_y"),
    ]
    output_keys = [
        Key("sol", scale=(5.74634e-03, 3.88433e-03)),
    ]

    download_FNO_dataset("Darcy_241", outdir="datasets/")
    invar_train, outvar_train = load_FNO_dataset(
        "datasets/Darcy_241/piececonst_r241_N1024_smooth1.hdf5",
        [k.name for k in input_keys],
        [k.name for k in output_keys],
        n_examples=cfg.custom.ntrain,
    )
    invar_test, outvar_test = load_FNO_dataset(
        "datasets/Darcy_241/piececonst_r241_N1024_smooth2.hdf5",
        [k.name for k in input_keys],
        [k.name for k in output_keys],
        n_examples=cfg.custom.ntest,
    )

    # add additional constraining values for darcy variable
    outvar_train["darcy"] = np.zeros_like(outvar_train["sol"])

    train_dataset = DictGridDataset(invar_train, outvar_train)
    test_dataset = DictGridDataset(invar_test, outvar_test)
```

Load additional gradients for the loss calculation

Loss for physics constraint

```
# Define FNO model
if cfg.custom.gradient_method == "hybrid":
    output_keys += [
        Key("sol", derivatives=[Key("x")]),
        Key("sol", derivatives=[Key("y")]),
    ]
model = instantiate_arch(
    input_keys=[input_keys[0]],
    output_keys=output_keys,
    cfg=cfg.arch.fno,
    domain_length=[1.0, 1.0],
)

# Make custom Darcy residual node for PINO
inputs = [
    "sol",
    "coeff",
    "Kcoeff_x",
    "Kcoeff_y",
]
if cfg.custom.gradient_method == "hybrid":
    inputs += [
        "sol_x",
        "sol_y",
    ]
darcy_node = Node(
    inputs=inputs,
    outputs=["darcy"],
    evaluate=Darcy(gradient_method=cfg.custom.gradient_method),
    name="Darcy Node",
)
nodes = model.make_nodes(name="FNO", jit=False) + [darcy_node]
```

Additional nodes for physics loss

# Darcy Flow – Physics Informed Neural Operators (PINO)

Code snippets and results and discussion

```
# make domain
domain = Domain()

# add constraints to domain
supervised = SupervisedGridConstraint(
    nodes=nodes,
    dataset=train_dataset,
    batch_size=cfg.batch_size.grid,
)
domain.add_constraint(supervised, "supervised")

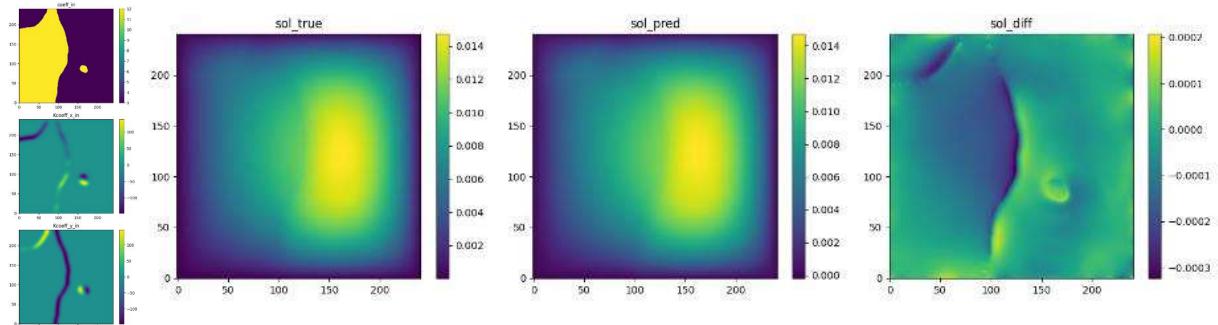
# add validator
val = GridValidator(
    nodes,
    dataset=test_dataset,
    batch_size=cfg.batch_size.validation,
    plotter=GridValidatorPlotter(n_examples=5),
    requires_grad=True,
)
domain.add_validator(val, "test")

# make solver
slv = Solver(cfg, domain)

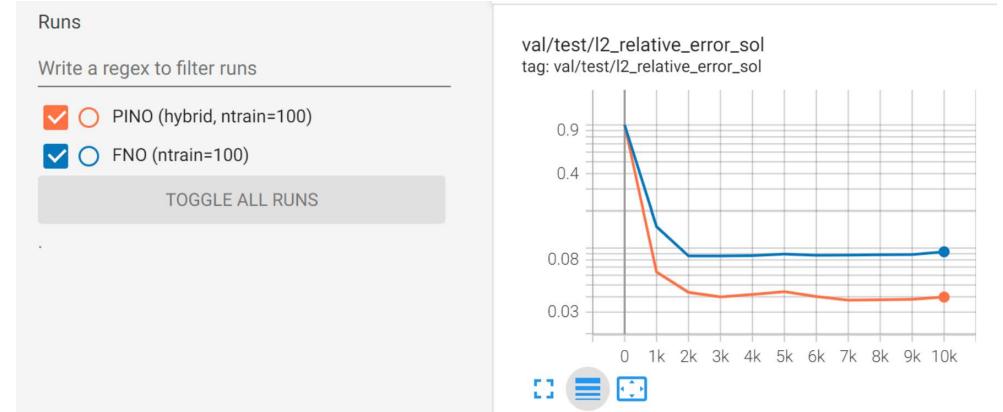
# start solver
slv.solve()
```

Constraints and Validators same as FNO example.

Without any changes to the core constraint/validator, experiment with different architectures with ease



Results. Left: Inputs to the model (Input permeability and its gradients), Right: Model prediction and its comparison with true solution



Benefit of PINO is seen in “small data” regimes where the PDE loss regularizes the model giving better prediction than FNO when trained with only 100 samples.

# Neural Operators in Modulus

## Conclusions

- Modulus supports several neural operators (also includes the famous DeepONets) that can easily be used out of the box.
- Modulus uses Hydra configuration for easily adjusting hyperparameters and even changing model architectures with minimal changes to Python code.
- Data-driven training can quickly be integrated into the Modulus framework.
- For more information on supported models and examples please see the [Modulus documentation](#).

# Modulus Features

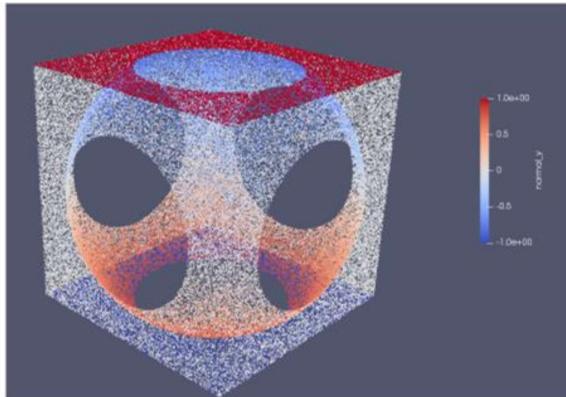
Constructive Solid Geometry Module – Fully parameterizable with SymPy and allows Boolean ops

- Allows to create complex geometries using a variety of primitives (torus, cones, ellipse, etc.)
- Supports functions like translate, rotate, scale, repeat, etc.

```
from modulus.geometry.csg.csg_3d import Sphere, Box
from modulus.plot_utils.vtk import var_to_polyvtk

# define geometry
t = Sphere((0,0,0), 1.2)
b = Box((-1,-1,-1), (1,1,1))
geo = b - t

# sample surface
surface_points = geo.sample_boundary(4096)
var_to_polyvtk(surface_points, 'surface_points')
```

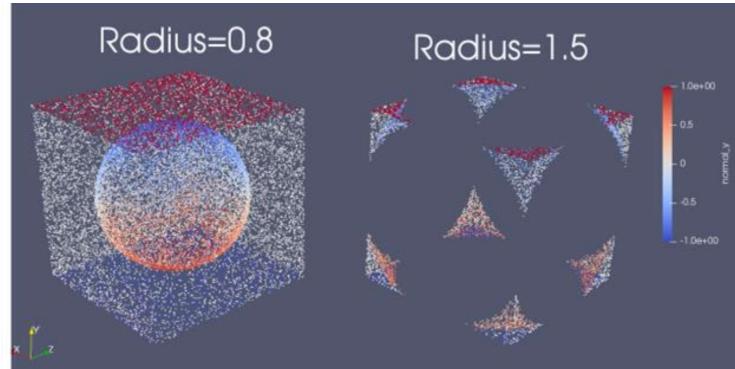


Boolean (Union, Intersection and, Subtraction) of different primitives within Modulus

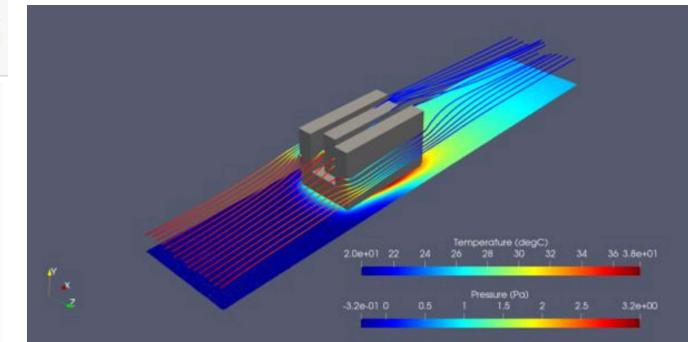
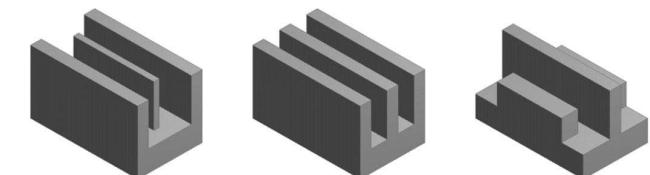
```
from modulus.geometry.csg.csg_3d import Sphere, Box
from modulus.plot_utils.vtk import var_to_polyvtk
from sympy import Symbol

# define geometry
radius = Symbol('radius')
radius_range = {radius: (1.5, 0.8)}
t = Sphere((0,0,0), radius)
b = Box((-1,-1,-1), (1,1,1))
geo = b - t

# sample surface
surface_points = geo.sample_boundary(16384, param_ranges=radius_range)
var_to_polyvtk(surface_points, 'surface_points')
```



Geometry parameterizable using SymPy

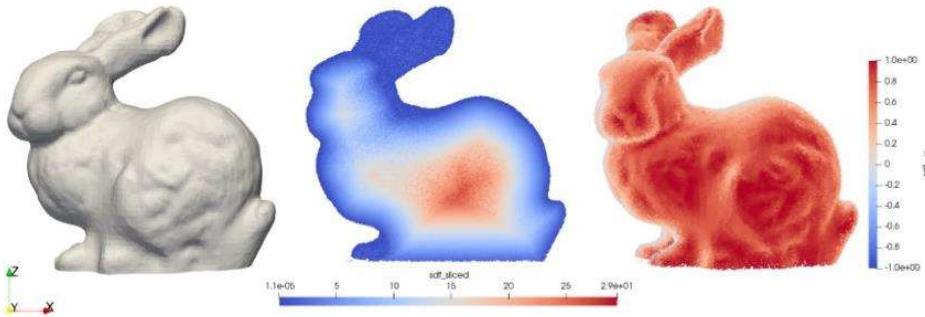


Example of a parameterized 3 Fin heatsink and solving the conjugate heat transfer problem.

## Modulus Features

PySDF library: Tessellated Geometry Module, OptiX for sampling and SDF

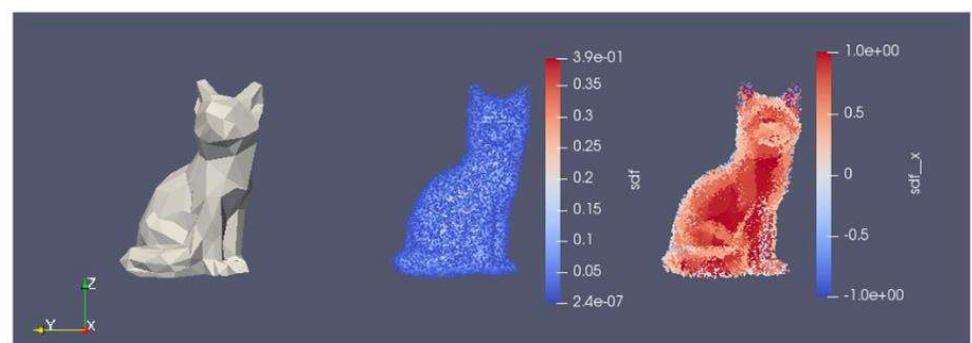
- Allows to import complex tessellated geometries.
- Uses ray tracing to compute SDF and its derivatives. Also computes surface normals.
- Once the geometry is imported, creates a point cloud for training.



```
from modulus.geometry.tessellation.tessellation import Tessellation
from modulus.plot_utils.vtk import var_to_polyvtk

# read stl
geo = Tessellation.from_stl("fox.stl", airtight=True)

# sample points in interior
interior_points = geo.sample_interior(40000, compute_distance_field=True)
var_to_polyvtk(interior_points, 'interior_points')
```



# Modulus Features and Advancements

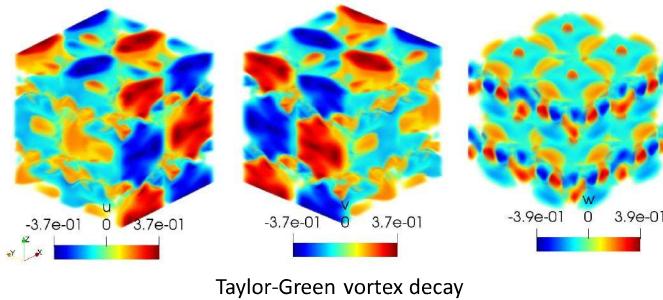
Physics types and solution of differential equations

Physics types:

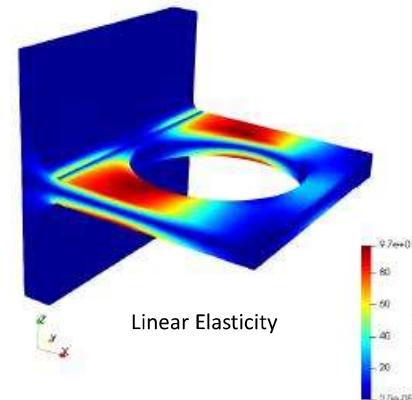
- Linear Elasticity (plane stress, plane strain and 3D)
- Fluid Mechanics
- Heat Transfer
- Coupled Fluid-Thermal
- Electromagnetics
- Wave propagation
- Examples of 2-eqn. turbulence (on fully developed channel case)

Solution of differential equations:

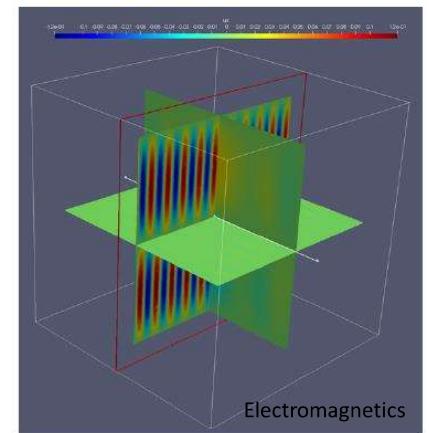
- Ordinary Differential Equations
- Differential (strong) Form
- Integral (weak) form of the PDEs



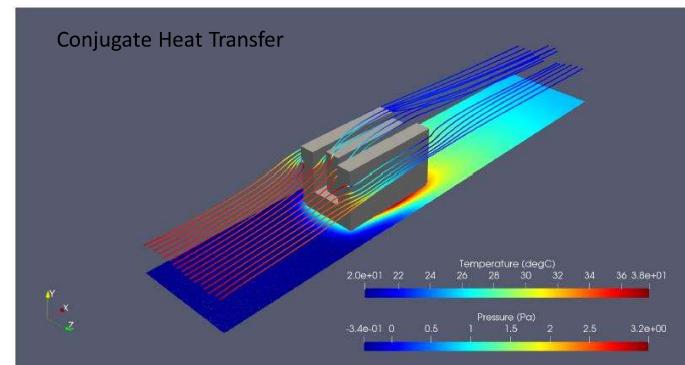
Taylor-Green vortex decay



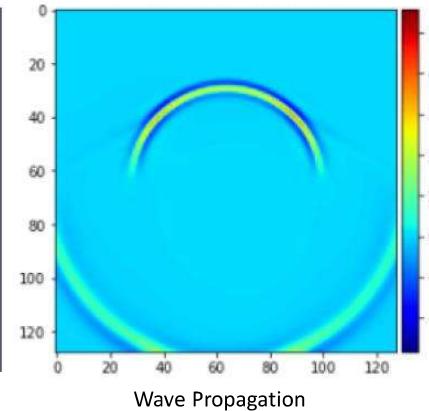
Linear Elasticity



Modulus t=600ms



Conjugate Heat Transfer



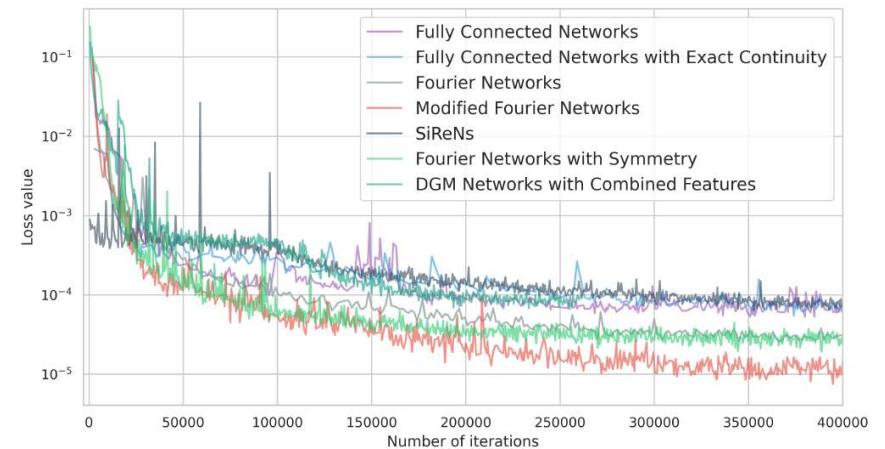
Wave Propagation

# Modulus Features and Advancements

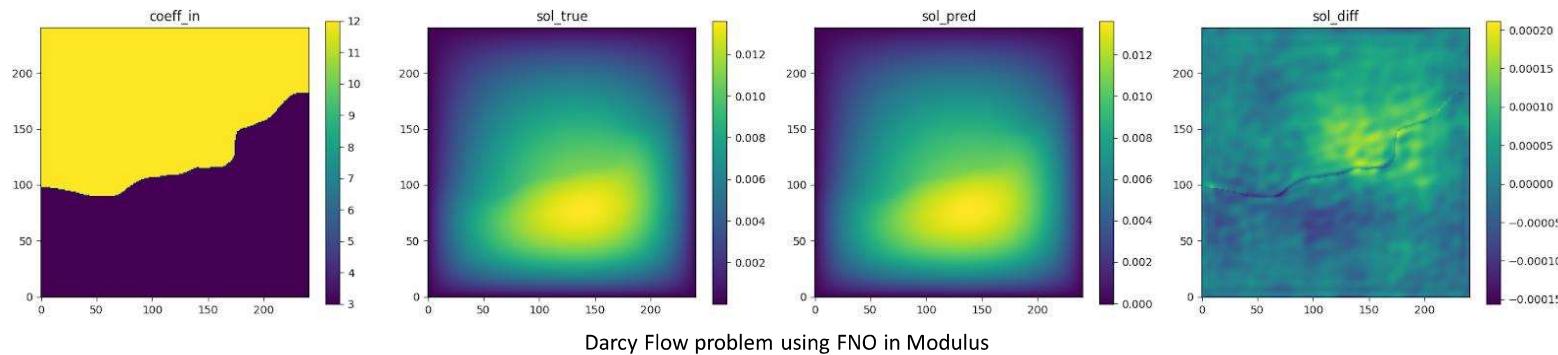
## Network Architectures

Several neural network architectures:

- Fully connected Network
- Fourier Feature Network
- Sinusoidal Representation Network (SiReN)
- Modified Fourier Network
- Deep Galerkin Method Network
- Modified Highway Network
- Multiplicative Filter Networks
- Operators:
  - Fourier Neural Operator (FNO)
  - Adaptive Fourier Neural Operator (AFNO)
  - Physics Informed Fourier Neural Operator (PINO)
  - DeepONet



Comparisons of various networks in Modulus applied to solve the flow over a heatsink



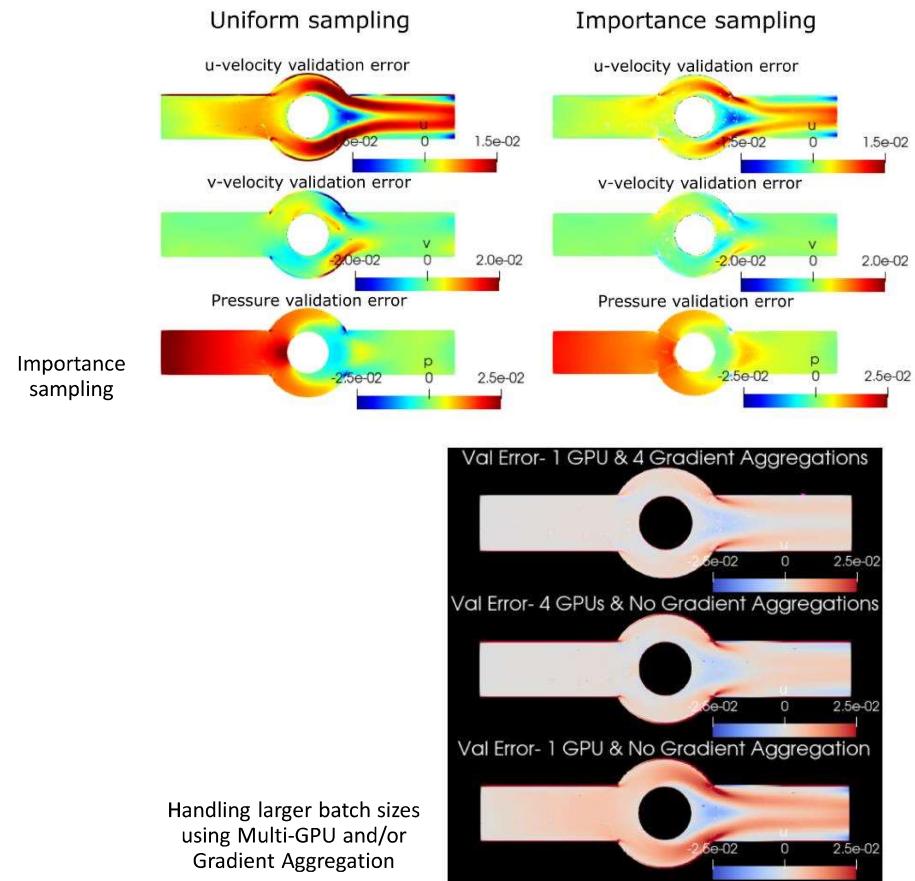
Darcy Flow problem using FNO in Modulus

# Modulus Features and Advancements

## Other advanced training features

Other Features include:

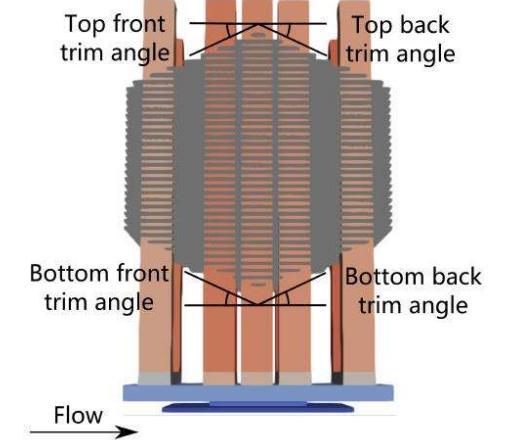
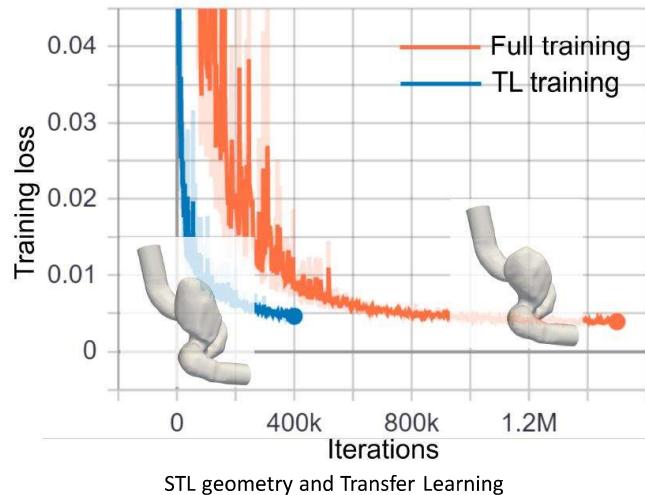
- Support for new optimizers
  - AdaHessian and up to 30+ optimizers (from PyTorch's `torch_optimizer` library)
- New algorithms for loss balancing
  - GradNorm, ReLoBRaLo, Soft Adapt, NTK, etc.
- Sobolev (gradient-enhanced) training
- Exact boundary condition imposition
- Global and local learning rate annealing
- Global adaptive activation functions
- Halton sequences for low-discrepancy point cloud creation
- Gradient Accumulation
- Time-stepping schemes for transient problems
- Temporal loss weighting and time marching for the continuous time approach
- Importance sampling
- Homoscedastic task uncertainty quantification for loss weighting
- Hydra Configs for easy hyper-parameter tuning



# Modulus Features and Advancements

## Other advanced training features

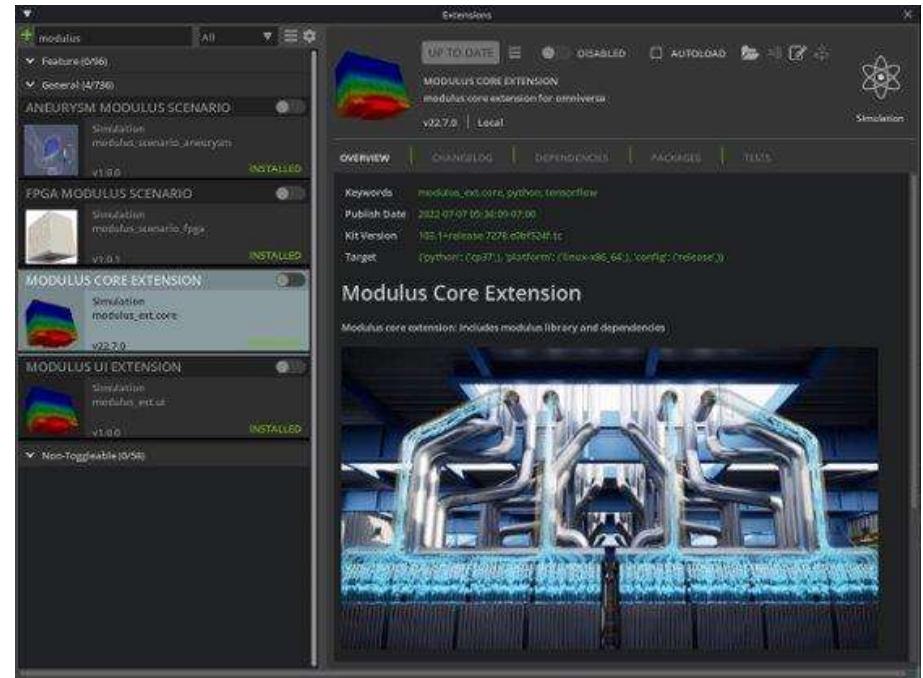
- APIs to automatically generate point clouds from Boolean compositions of geometry primitives or import point cloud for complex geometry (e.g., STL files)
- Parameterized system representation that solves several configurations concurrently for analytical geometry using Modulus CSG module
- Transfer learning for efficient surrogate-based parameterization of STL and constructive solid geometries



# Omniverse Extension for Modulus



- NVIDIA Omniverse™ is an easily extensible platform for 3D design collaboration and scalable multi-GPU, real-time, true-to-reality simulation. Omniverse revolutionizes the way we create and develop as individuals and work together as teams, bringing more creative possibilities and efficiency to 3D creators, developers and enterprises.
- Omniverse extension for Modulus enables real-time virtual-world simulation and full-design fidelity visualization. The built-in pipeline can be used for common visualizations such as streamlines and iso-surfaces for the outputs of the Modulus model. Another key feature is being able to visualize and analyze the high-fidelity simulation output in near real time as the design parameters are varied.
- For more information, please refer [Modulus Extension Docs](#)



# Modulus Resources

- Download Now: <https://developer.nvidia.com/modulus-downloads>
- Webpage: <https://developer.nvidia.com/modulus>
- Documentation: <https://sw-docs-dgx-station.nvidia.com/deeplearning/modulus/index.html>
- Developer Forum: <https://forums.developer.nvidia.com/c/physics-simulation>
- Demos:
  - [Accelerating Extreme Weather Prediction with FourCastNet](#)
  - [Siemens Energy HRSG Digital Twin Simulation Using NVIDIA Modulus and Omniverse](#)
  - [Accelerating Scientific & Engineering Simulation Workflows with AI](#)
  - [Flow Physics Quantification in an Aneurysm Using NVIDIA Modulus](#)
- Blogs:
  - [AI and Machine Learning in Physics](#)
  - [Using NVIDIA Modulus and Omniverse Wind Farm Digital Twin for Siemens Gamesa \(using NVIDIA Modulus and Omniverse\)](#)
  - [Siemens Energy Taps NVIDIA to Develop Industrial Digital Twin of Power Plant in Omniverse and Modulus](#)
  - [Using Hybrid Physics-Informed Neural Networks for Digital Twins in Prognosis and Health Management](#)
  - [Using Physics-Informed Deep Learning for Transport in Porous Media](#)

# Modulus Resources

- Papers:
  - [NVIDIA SimNet™: An AI-Accelerated Multi-Physics Simulation Framework](#)
  - [Physics-Informed Machine Learning and Uncertainty Quantification for Mechanics of Heterogeneous Materials](#)
  - [Physics Informed RNN-DCT Networks for Time-Dependent Partial Differential Equations](#)
- Talks:
  - [A Novel Framework for Physics based Machine Learning in Science & Engineering Domains](#)
  - [Accelerate HPC Simulations at Scale with Physics-informed Neural Networks and NVIDIA GPUs on AWS](#)
  - [Toward Developing High Reynolds Number, Compressible, Reacting Flows in Modulus](#)
  - [Physics-informed Neural Networks for Wave Propagation Using NVIDIA Modulus](#)
  - [Uncertainty Quantification for Transport in Porous Media Using Parameterized Physics Informed Neural Networks](#)
  - [Using Physics-Informed Neural Networks and Modulus to Accelerate Product Development](#)
  - [Physics-Informed Neural Networks for Mechanics of Heterogenous Media](#)
  - [Hybrid Physics-Informed Neural Networks for Digital Twin in Prognosis and Health Management](#)
  - [Physics-Informed Neural Network for Flow and Transport in Porous Media](#)
  - [AI-Accelerated Computational Science and Engineering Using Physics-Based Neural Networks](#)
  - [Cumulative Damage Models, Hybrid-Physics-Informed Neural Networks, and Digital TwinsWWCumulative Damage Models, Hybrid-Physics-Informed Neural Networks, and Digital Twins](#)
  - [NVIDIA Modulus - Accelerating Scientific & Engineering Simulation workflows with AI](#)
- Podcast:
  - [AI Physics Simulation Toolkit SimNet & PINNs](#)

