End-to-End AI for Science Bootcamp

OpenACC
More Science, Less Programming

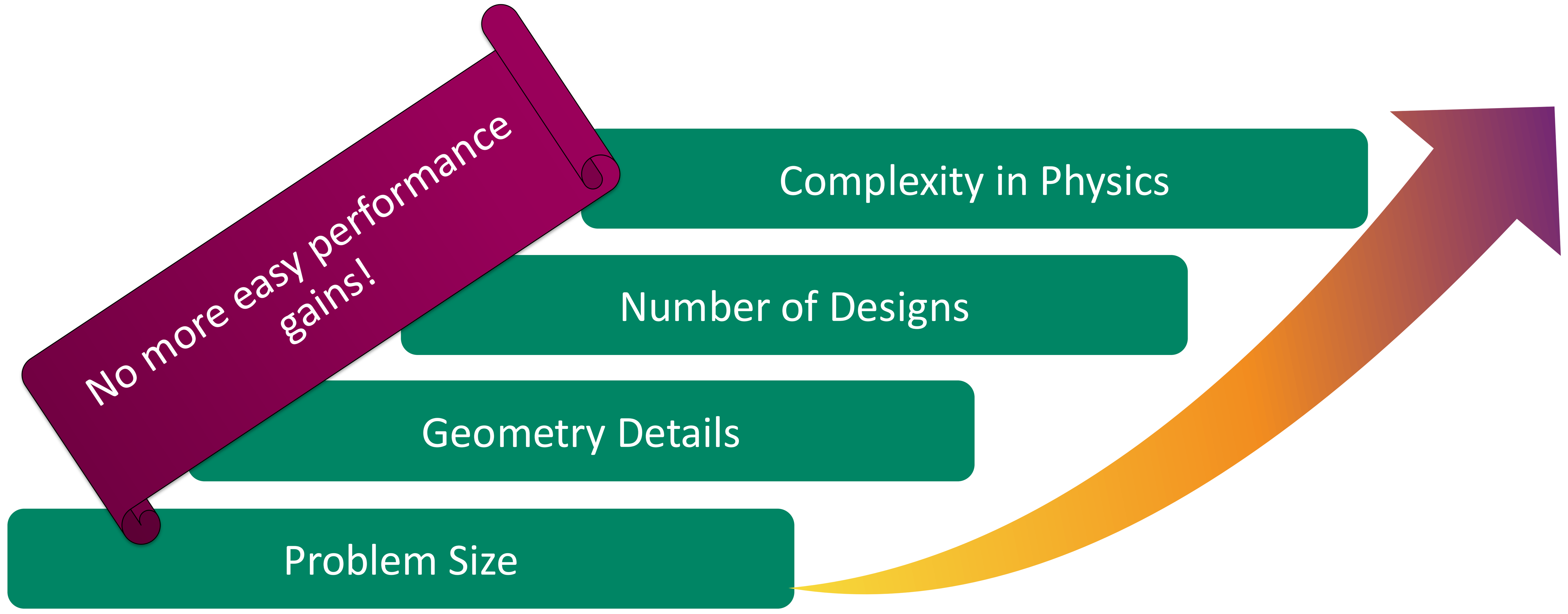
OPEN HACKATHONS

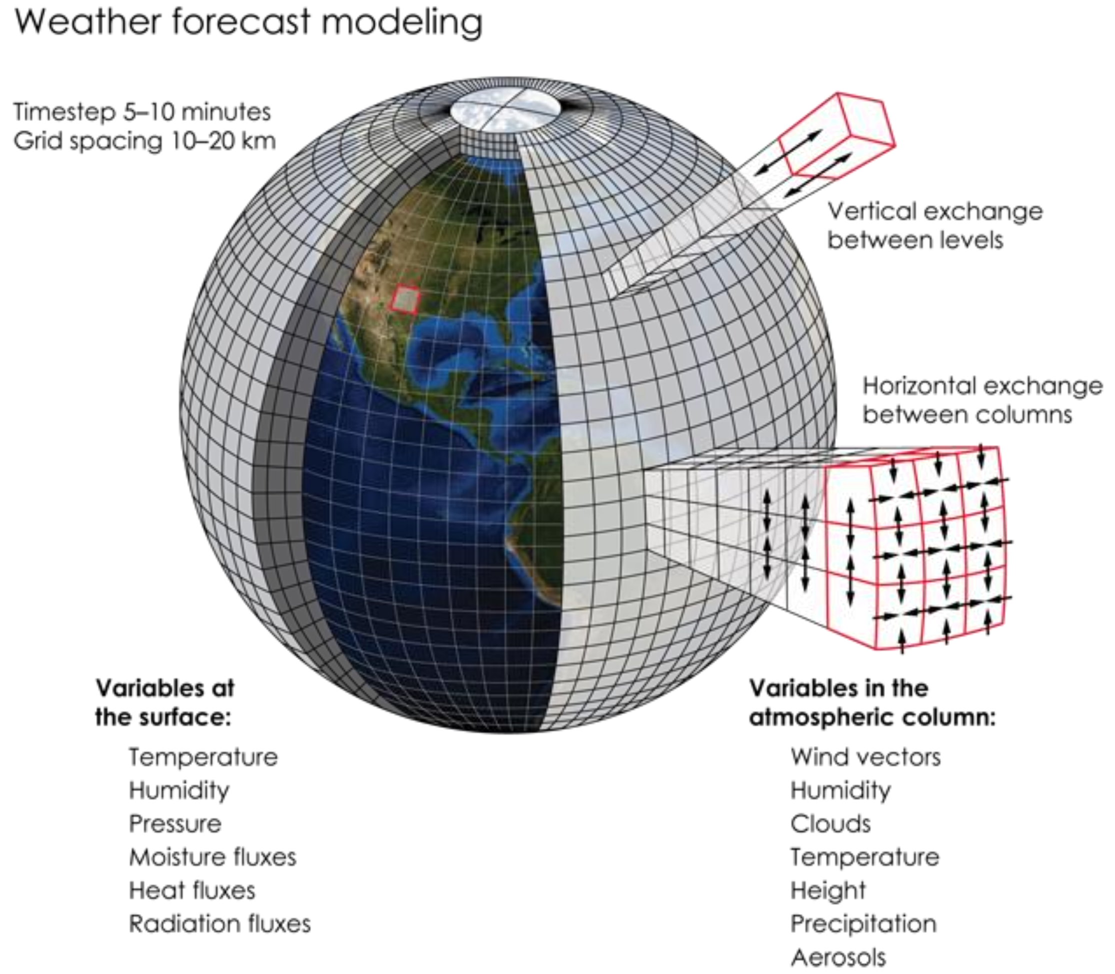
Saturating performance in traditional HPC

Simulations are getting larger and more complex

Traditional solution methods are:

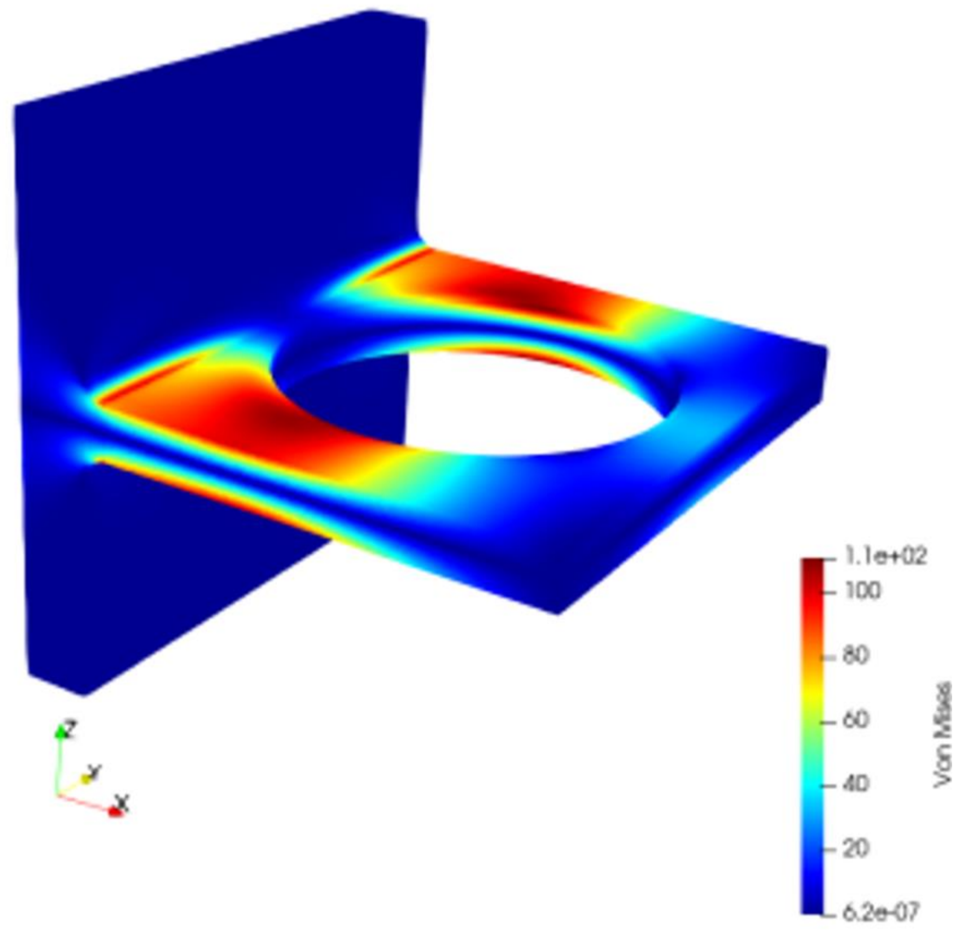
- Computationally Expensive
- Plagued by Domain Discretization Techniques
- Not suitable for Data-assimilation or Inverse problems





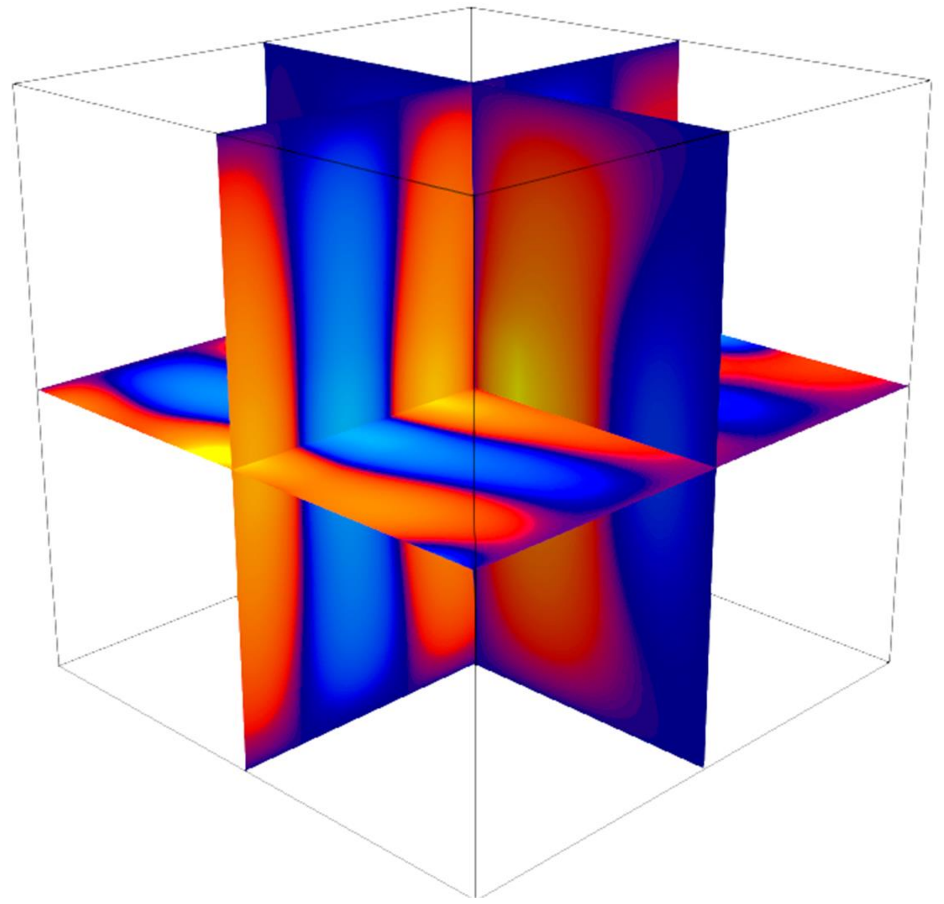
Atmospheric flows

Finite Volume Methods, Sub-grid scale modeling



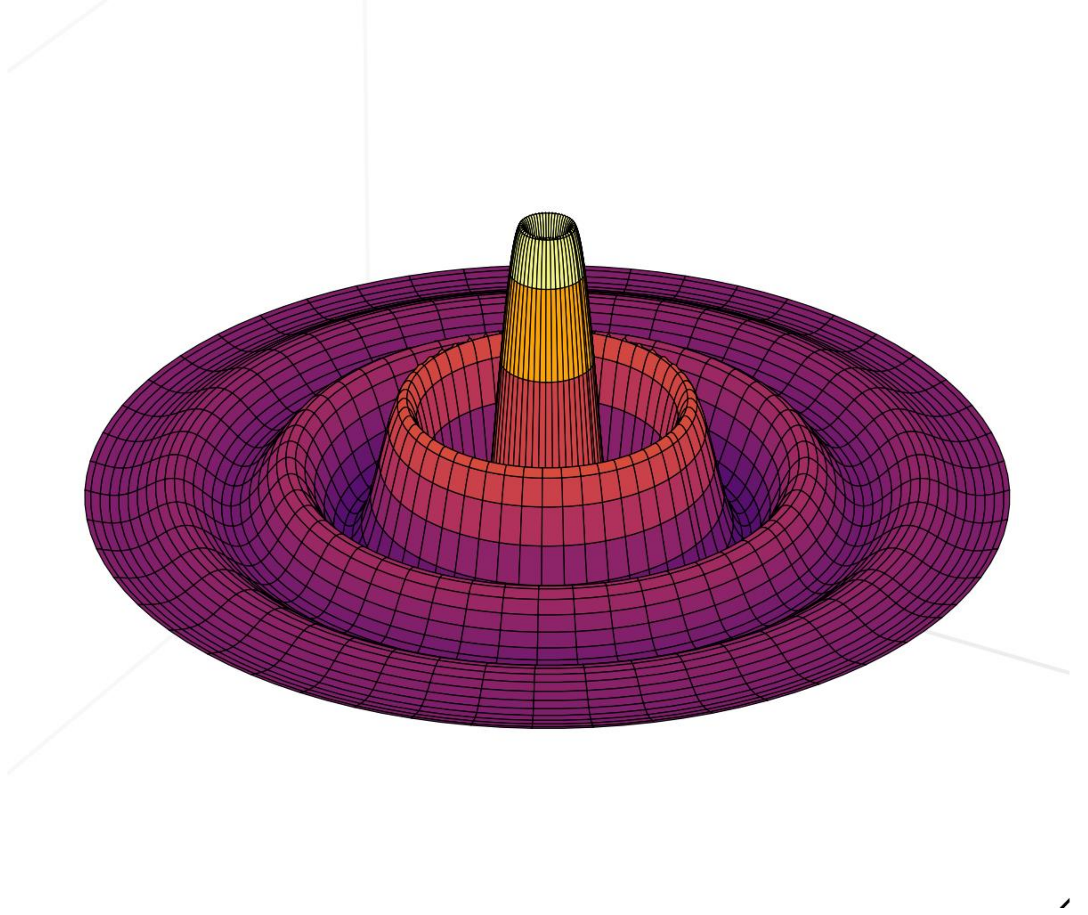
Structural Mechanics

Finite Element Methods



Electromagnetics

Finite Element and Frequency domain methods



Vibrations / Acoustics

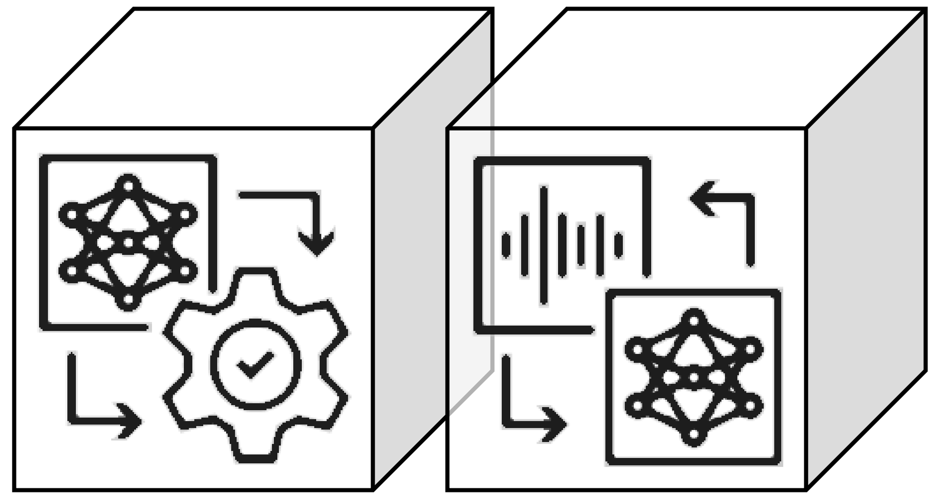
Finite Difference Methods

OpenACC OPEN HACKATHONS

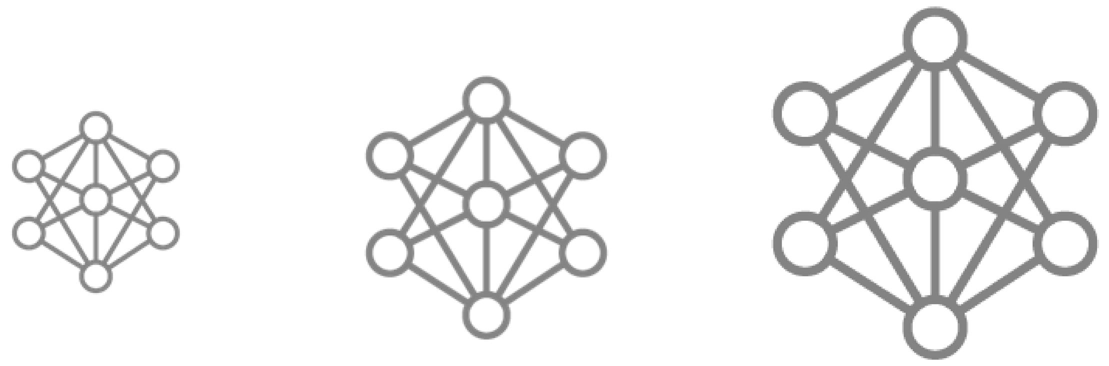
Multiple ways to incorporate AI for Scientific Research and Discovery

NeMo : Generative AI

NeMo Framework



Pretrained and Community models from NGC or HuggingFace



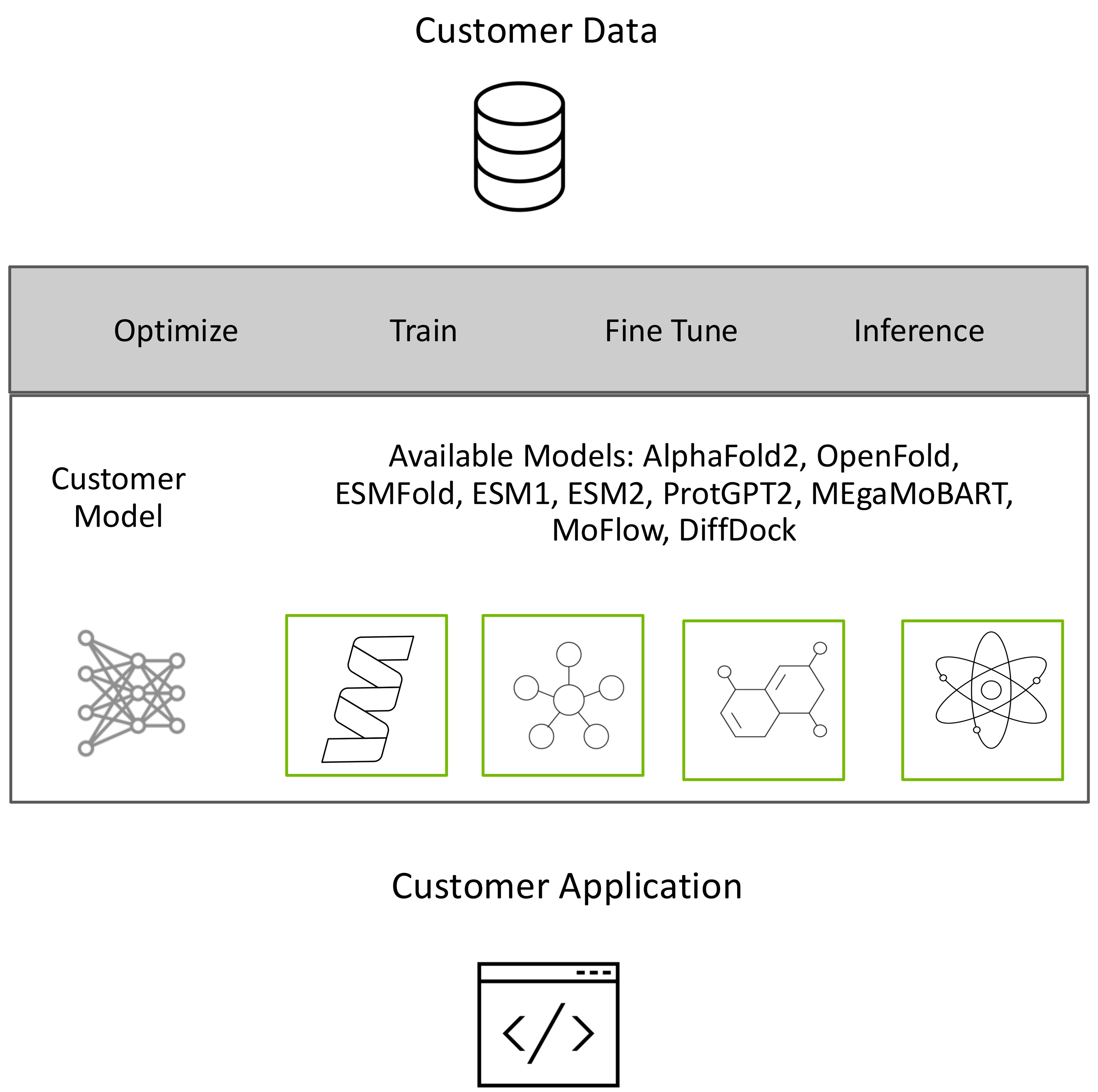
Early development techniques for researchers/developers

Nemo Inform

Nemo Guardrails

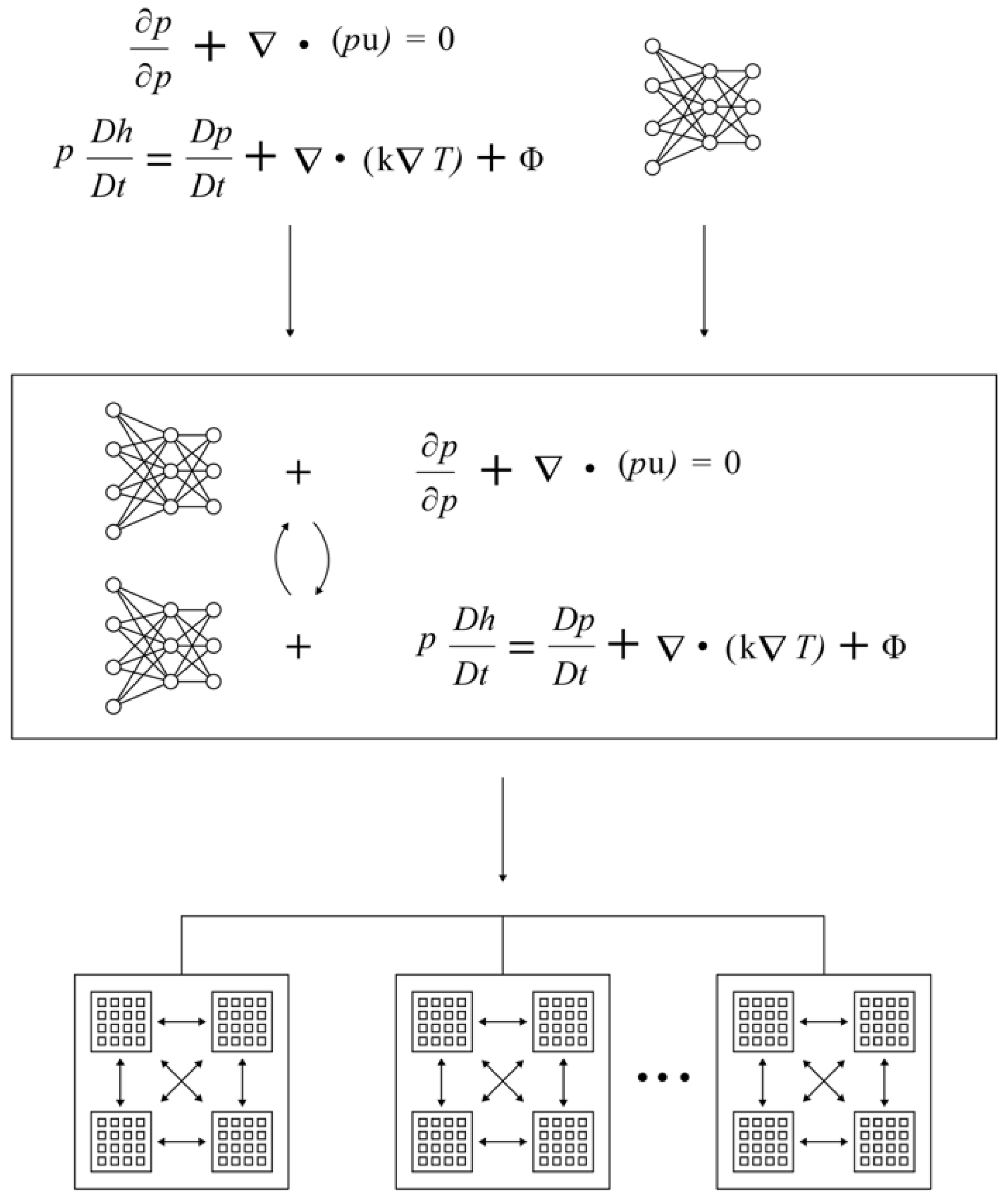
Nemo SteerLM

BioNeMo : Drug Discovery



A Cloud Managed Service for Customize and Run Generative AI for Computer Aided Drug Discovery

PhysicsNeMo : Physics-Based ML

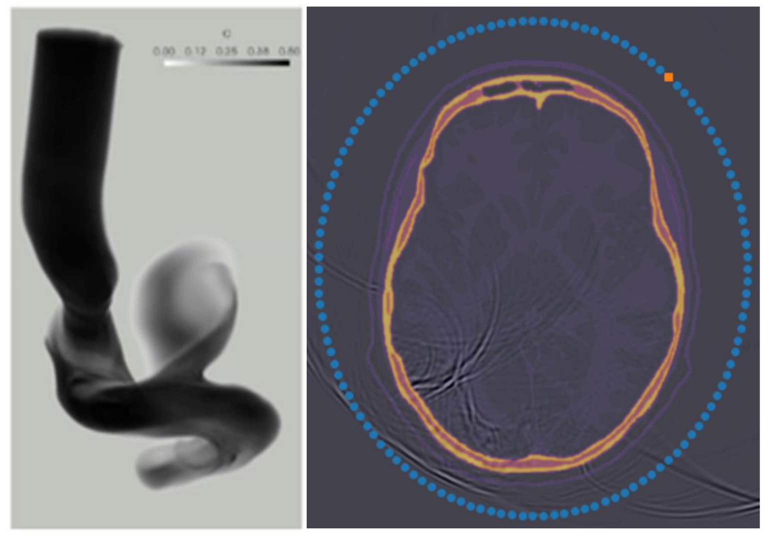


Training neural networks using both data and the governing equations

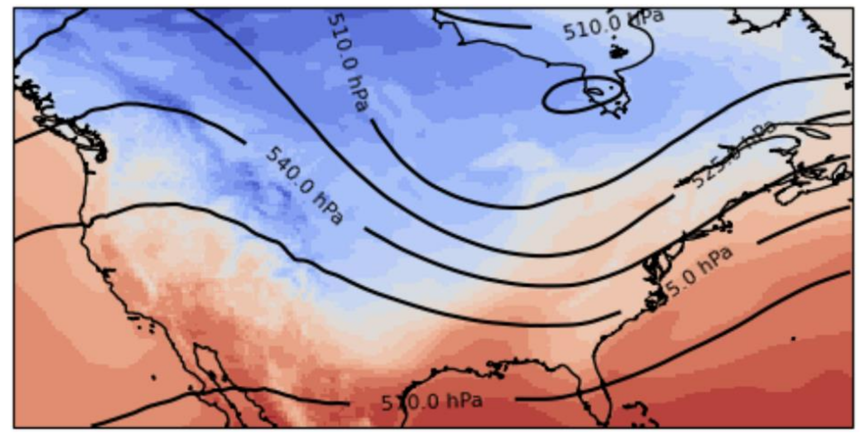
Using AI in Engineering and Science

Use data and governing equations to gather insight

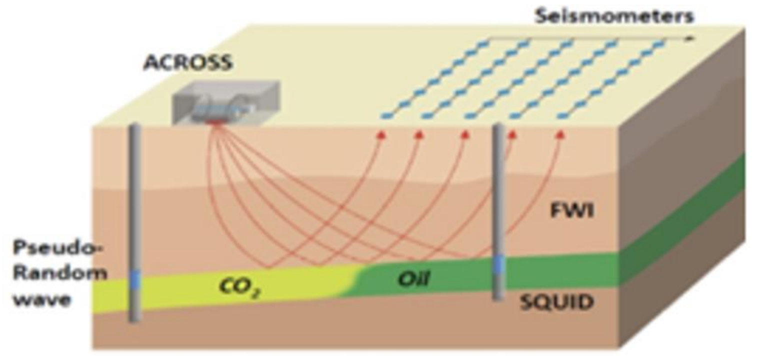
Inverse and Data Assimilation



Medical Imaging

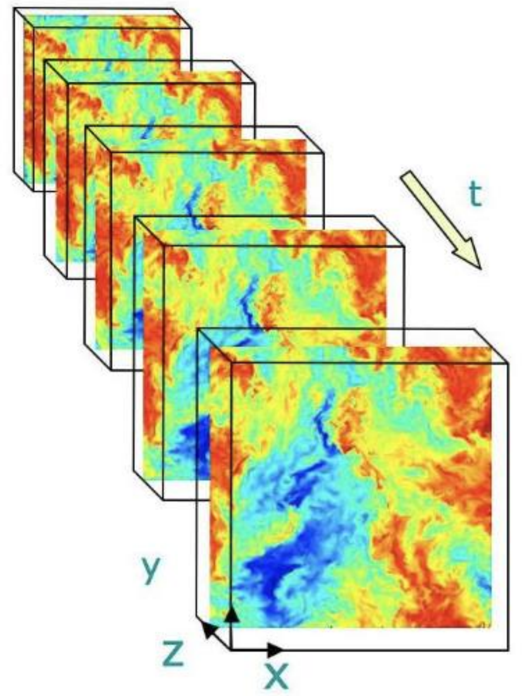


Weather & Climate

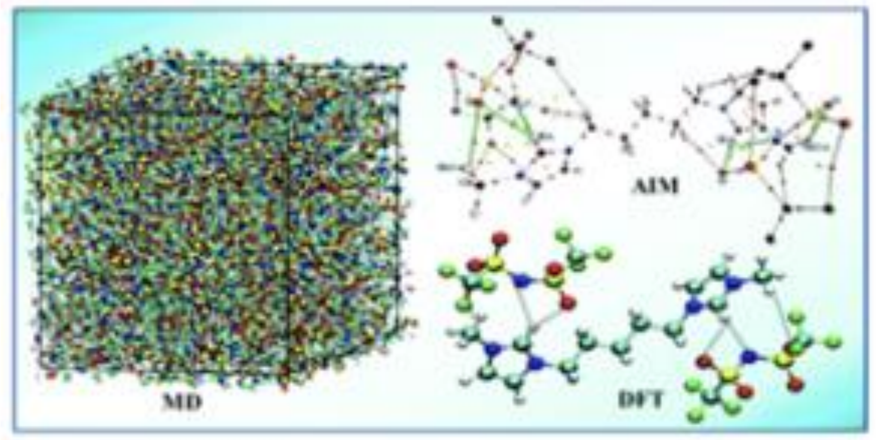


Oil & Gas

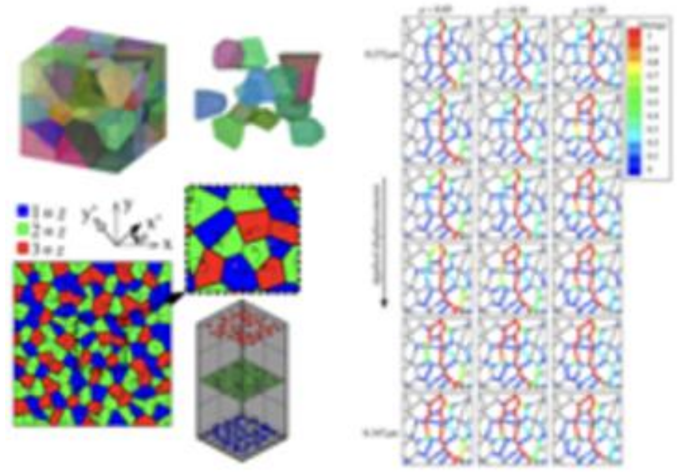
Parameterized Solutions



Turbulence



Molecular Dynamics

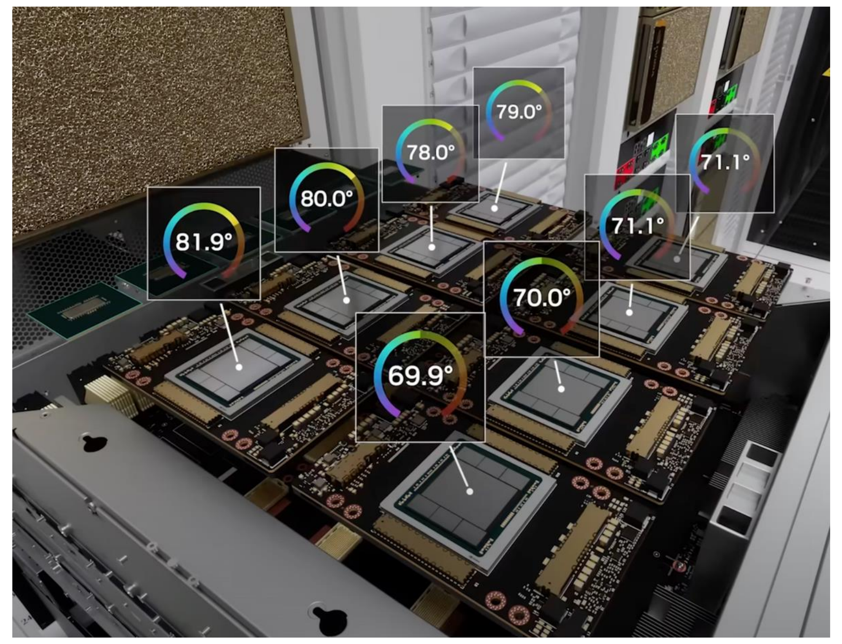


Micro-mechanical Material Model

Operational Control / Real-time

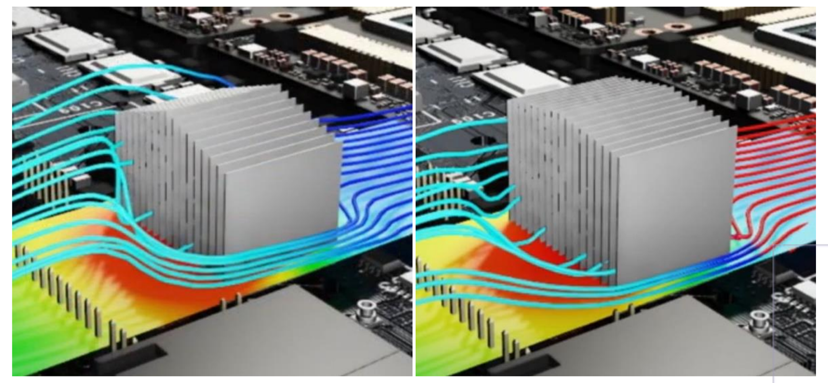
Robotics

Digital Twins



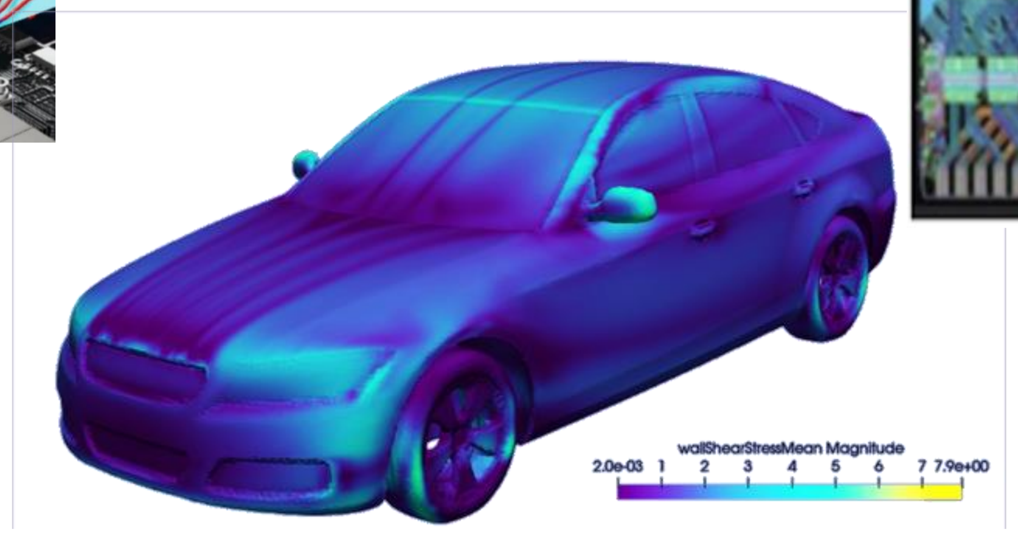
Autonomous Ride & Handling

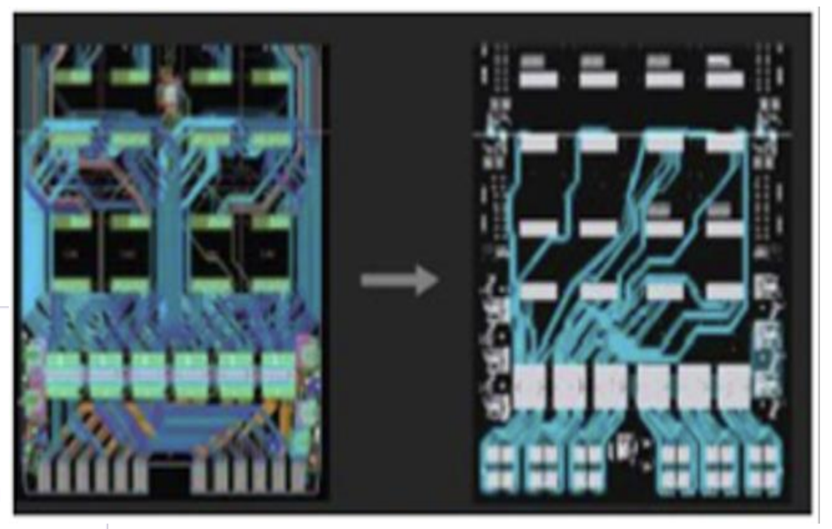
Digital Design and Manufacturing



Heatsinks

Aerodynamics





Circuit Design

Physics & Data – Little to no gain from Traditional Solver

Physics – Traditional Solver (Speed is a limitation)

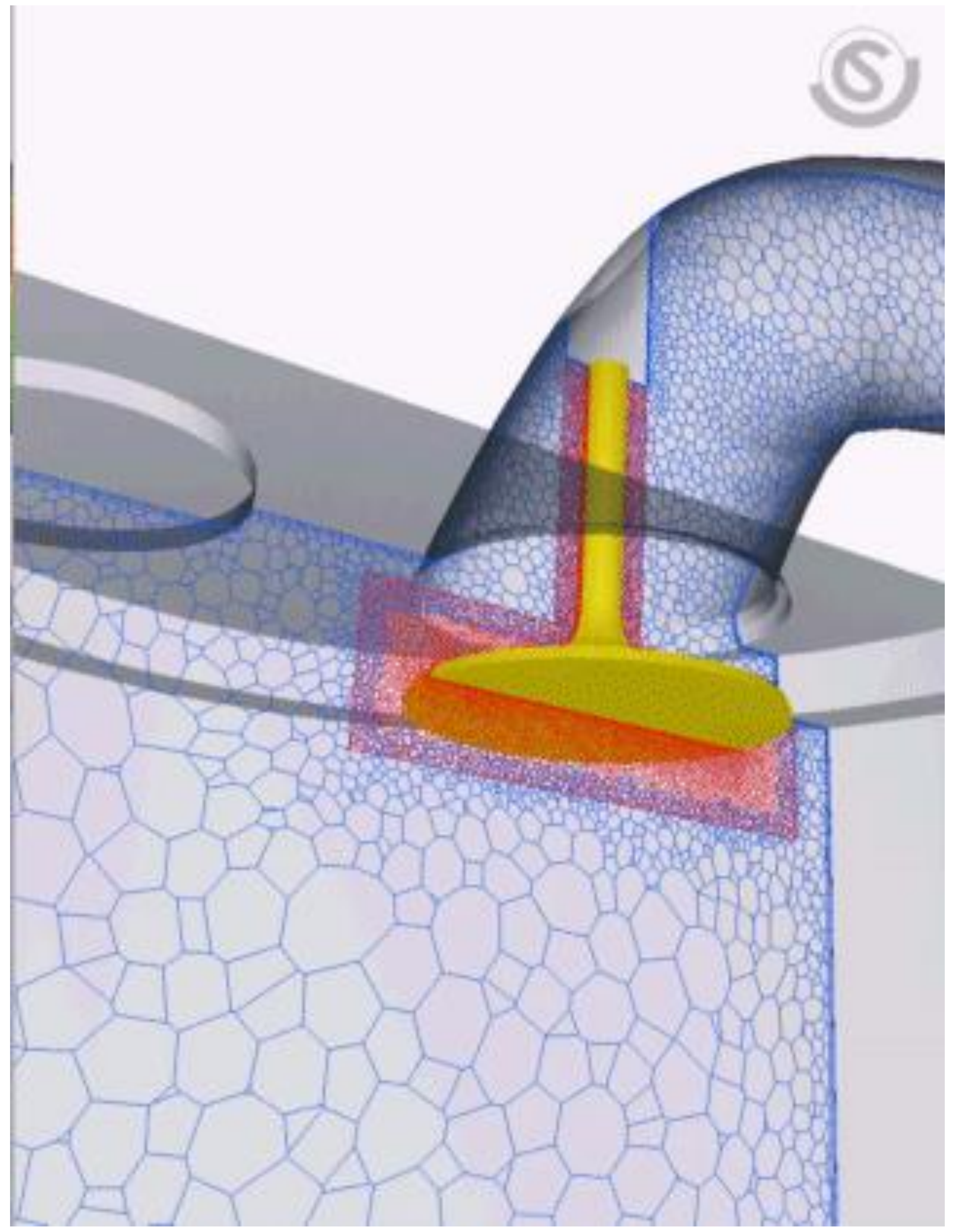
OpenACC

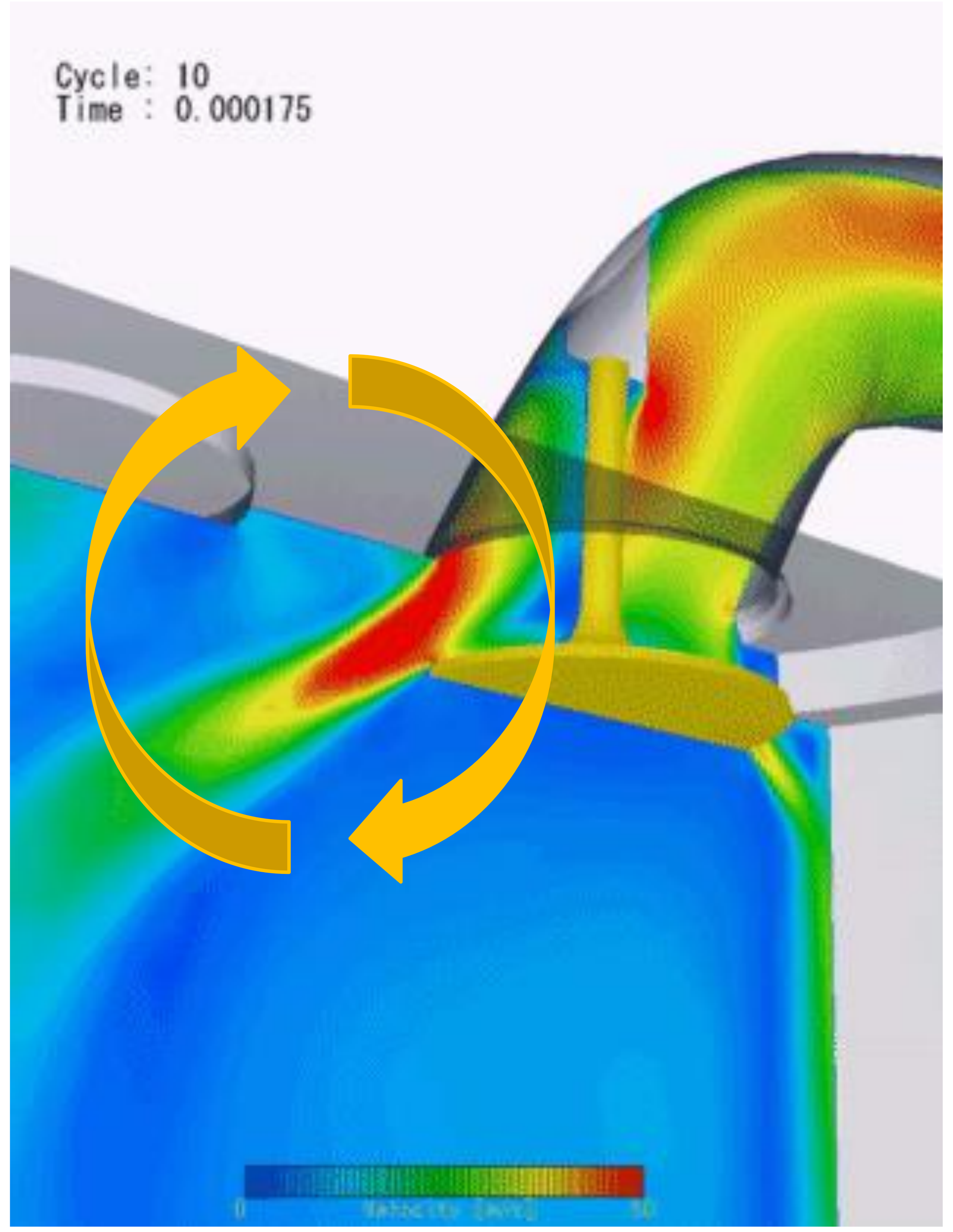
OPEN HACKATHONS

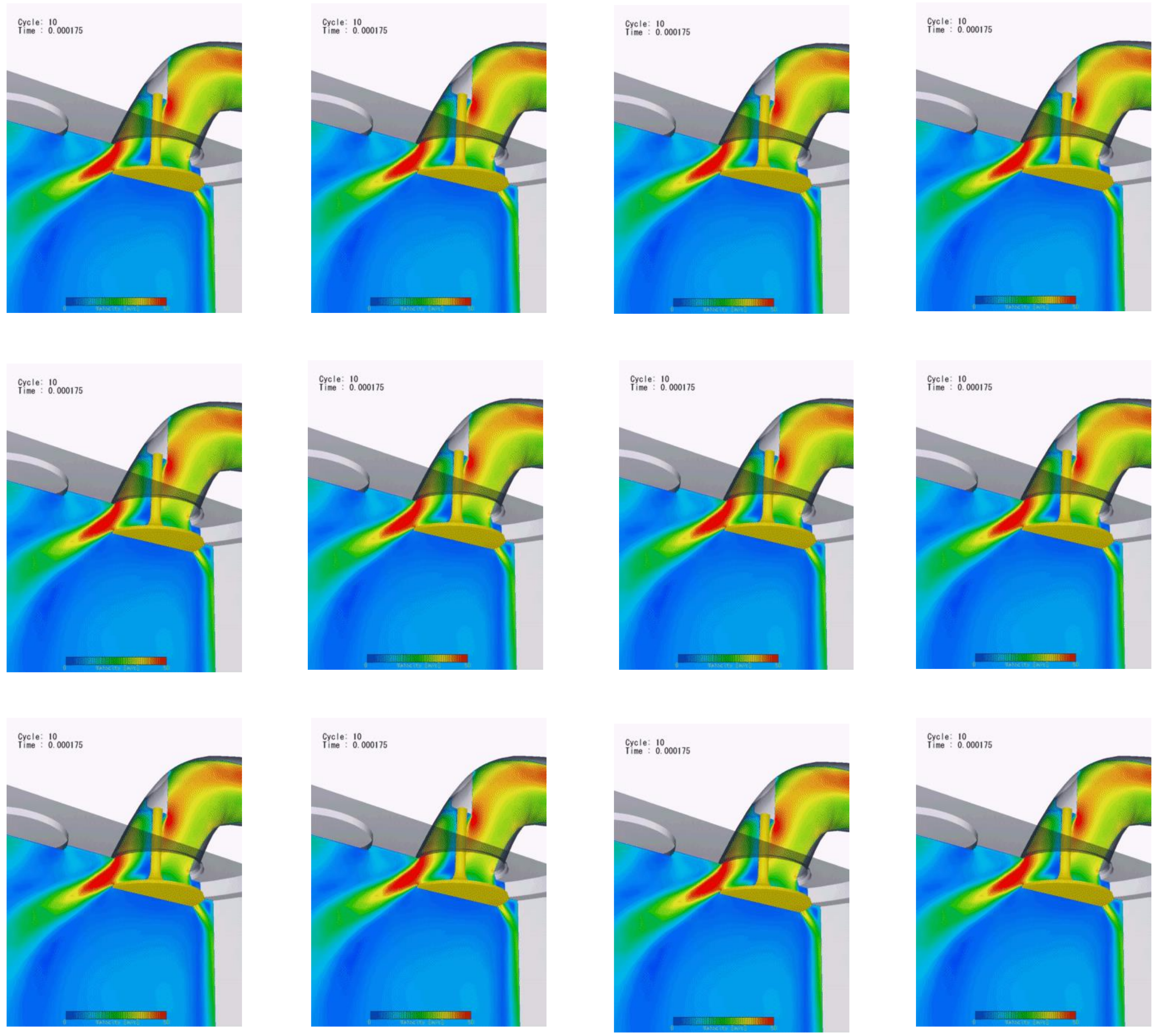
Using AI in Engineering and Science

Use data and governing equations to gather insight

- Like other domains that see the disruption due to AI
- Using AI in simulations unleashes parallelism, real-time outputs, inverse modeling capabilities and generative design







Real-time analysis of multiple scenarios

Agenda

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

PhysicsNeMo framework: Overview

Framework to build and customize Physics-ML models

Multi-domain support

Build physics-ml models for CFD, Heat Transfer, Structural, Electromagnetics, Molecular Dynamics

Optimized Training

Accelerate training and throughput by parallelizing the model and the training data across multi-node.

SOTA Model Architectures

Easily explore physics-ml model architectures – Neural Operators, PINNs, GNNs, Diffusion Models.

Support

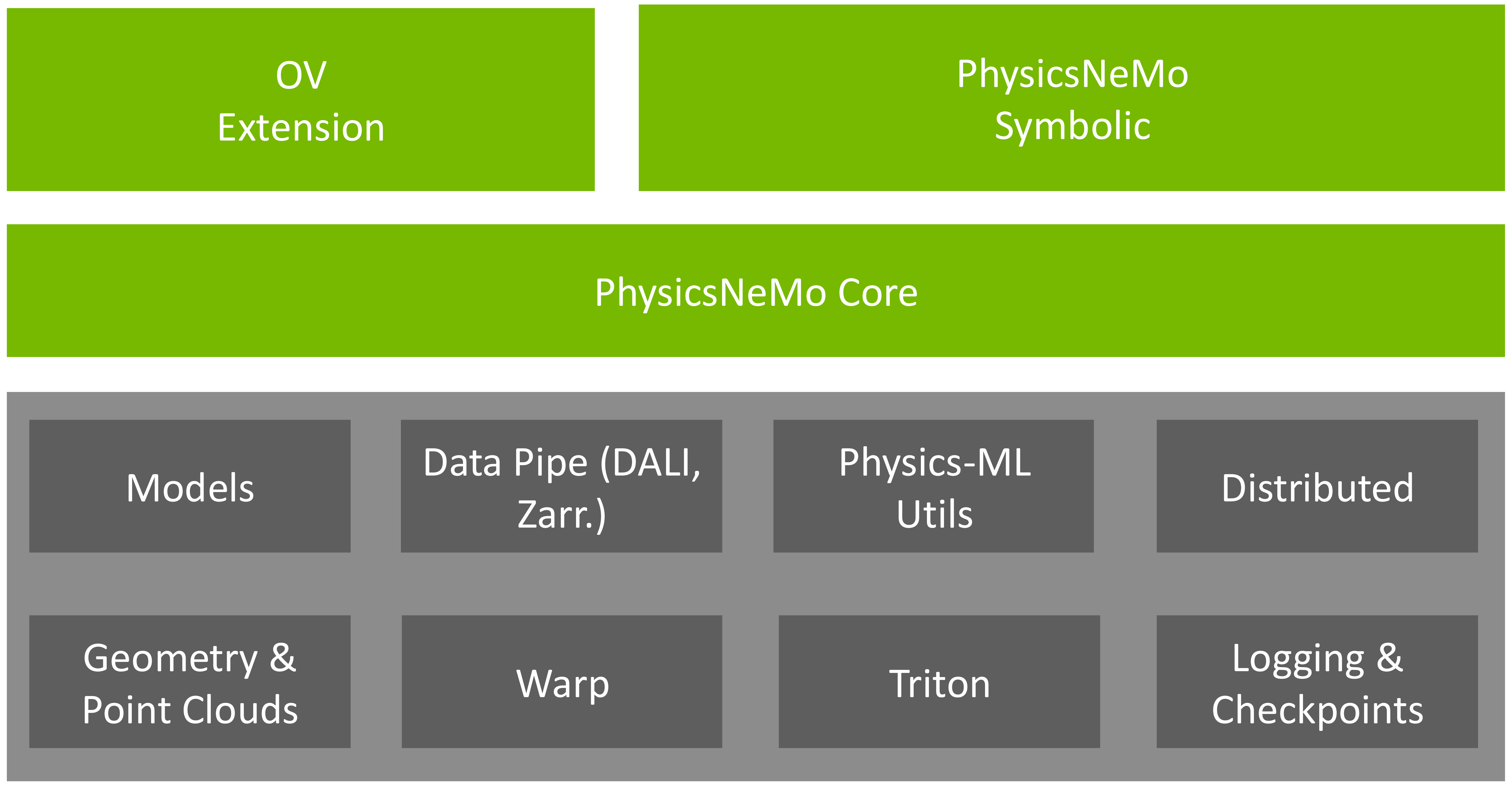
NVIDIA AI Enterprise and experts by your side to keep projects on track

General Availability – Part of NV AIE (Starting NV AIE 4.0)

PhysicsNeMo framework: Software stack and accessibility

Modular architecture to support domain experts as well as seasoned developers

- PhysicsNeMo Core provides fundamental and optimized implementations of data pipes, layers, models and utilities to setup distributed training pipelines
- PhysicsNeMo Sym provides abstractions to setup physics ML training with features like geometry module, PDEs, gradient utilities and optimized training loop
- OV extension allows easy export of models to NVIDIA Omniverse for visualization
- General availability via PyPi, NGC Container registry and open source on GitHub.

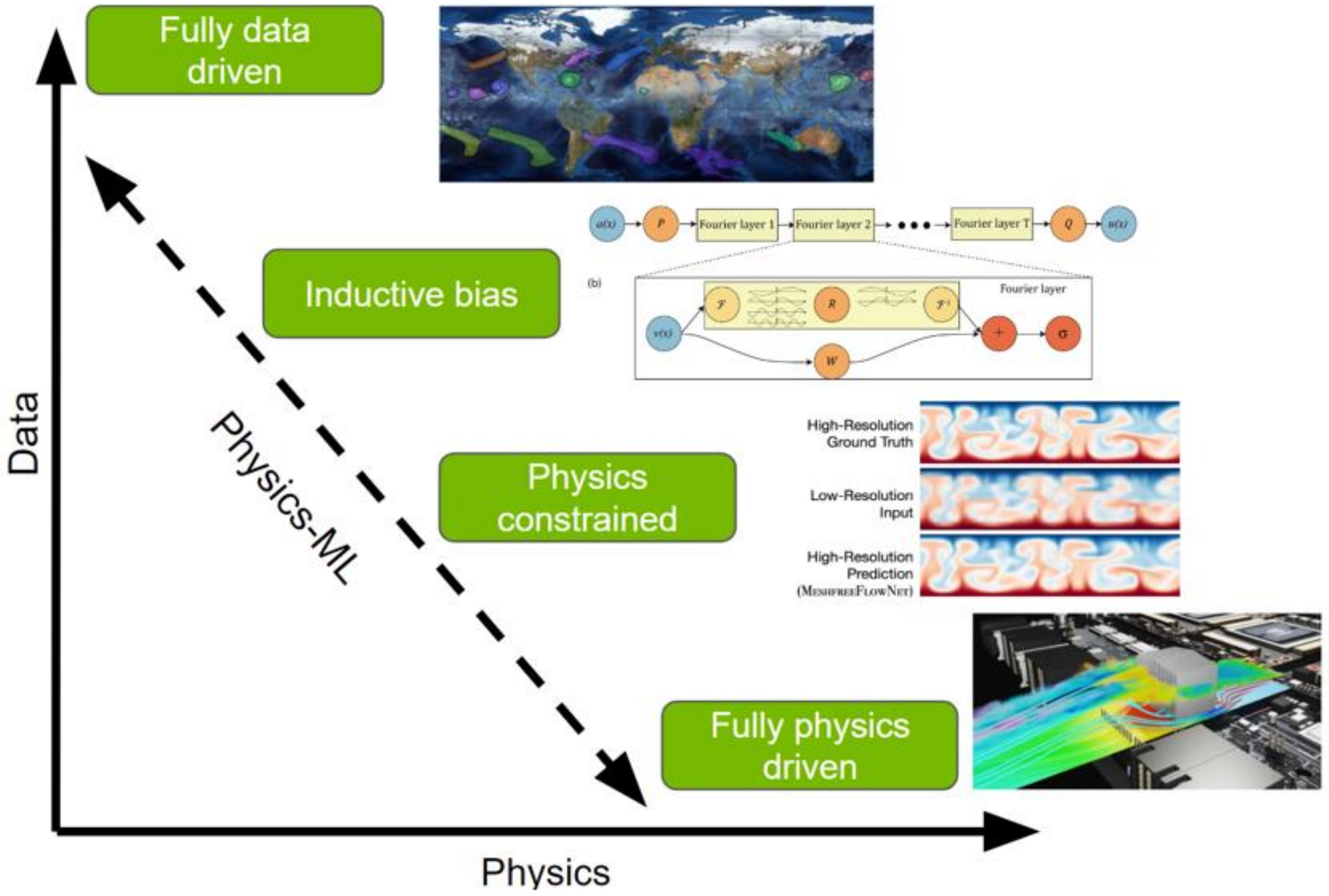


PhysicsNeMo framework: Open-Source AI for Physics-based ML

Novel NN architectures

- PhysicsNeMo Model Zoo - Diverse Physics-ML approaches:
  - Fully Physics driven AI models
  - Fully data driven AI models
  - Hybrid (data + Physics) AI models
- Neural Operators:
  - Fourier Neural Operator family (FNO, AFNO, PINO)
  - DeepONet
- GNNs:
  - GraphCast
  - MeshGraphNet …
- Diffusion Models:
  - DDPM++
  - NCSN++
  - ADM …
- PDE informed Neural Networks:
  - Fourier Feature Network
  - Spatial-temporal Fourier Feature Networks
  - SIREN Net …

- Bringing novel AI architectures that have demonstrated success for engineering and science problems
- Using case studies as reference starting points



How does PhysicsNeMo compliment PyTorch?

Features that can aid data and/or physics driven problems

Data & Physics oriented utils

- Performance Enhancements
CUDA graphs, kernel fusion, JIT compilation, data parallelism, model parallel, etc.
- Pre-built Network Architectures
Diffusion Models, Neural Hash Encoding, Neural Operators, Graph Networks, DCT-RNN, several variants of MLPs etc.
- Hydra Configuration
Hyper-parameter tuning and customization
- Data Pipeline
For very large data-driven problems using Zarr, NVComp, GDS
- Data and Inference Tools
Pre-defined datasets for common data formats (VTK, HDF5, …). Model export functions to TensorRT and Triton
- Integration with Other Products
Omniverse, PySDF, NVFuser, Triton, Tensor RT, DALI, Warp, etc.

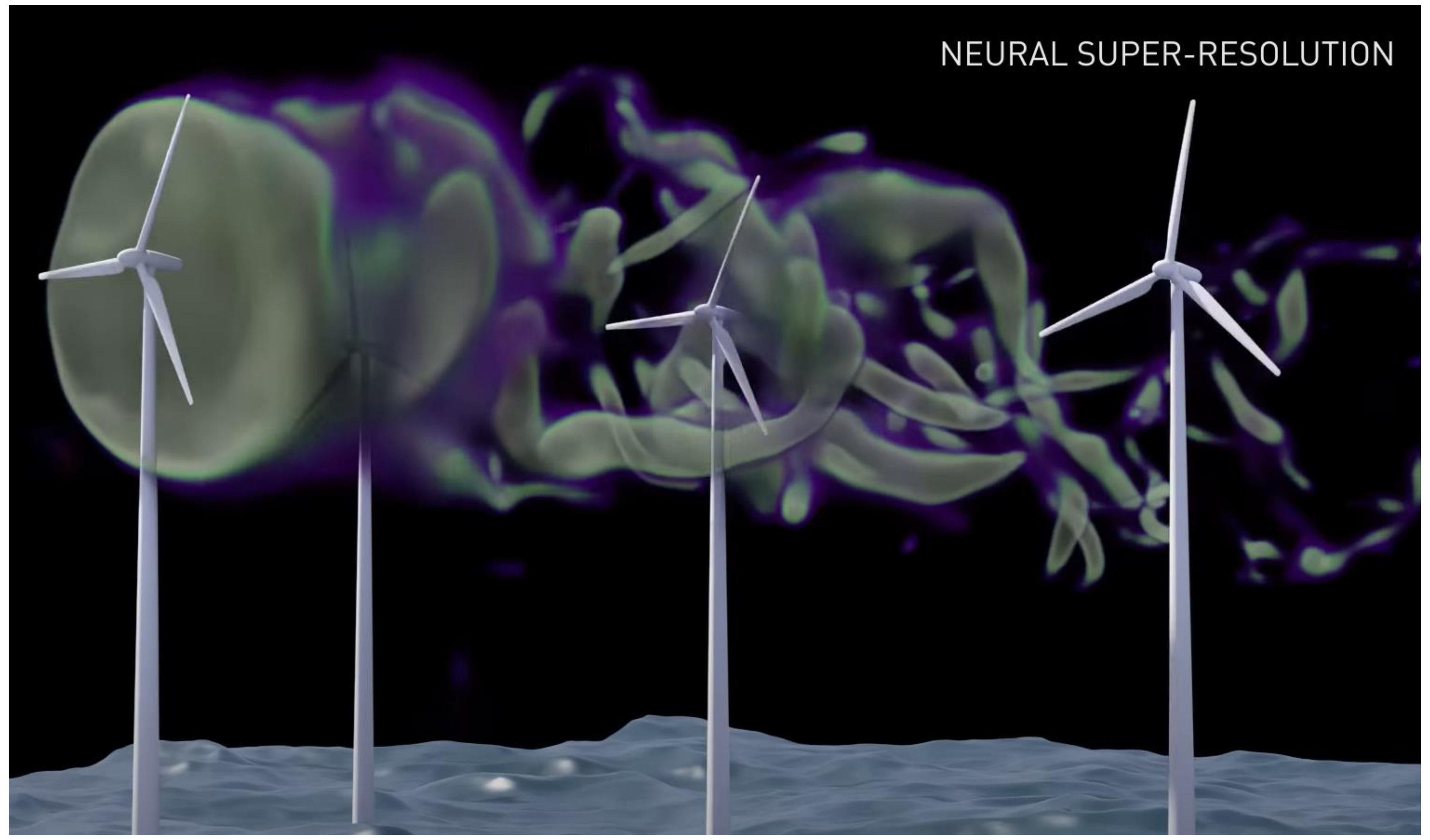
Physics oriented utils

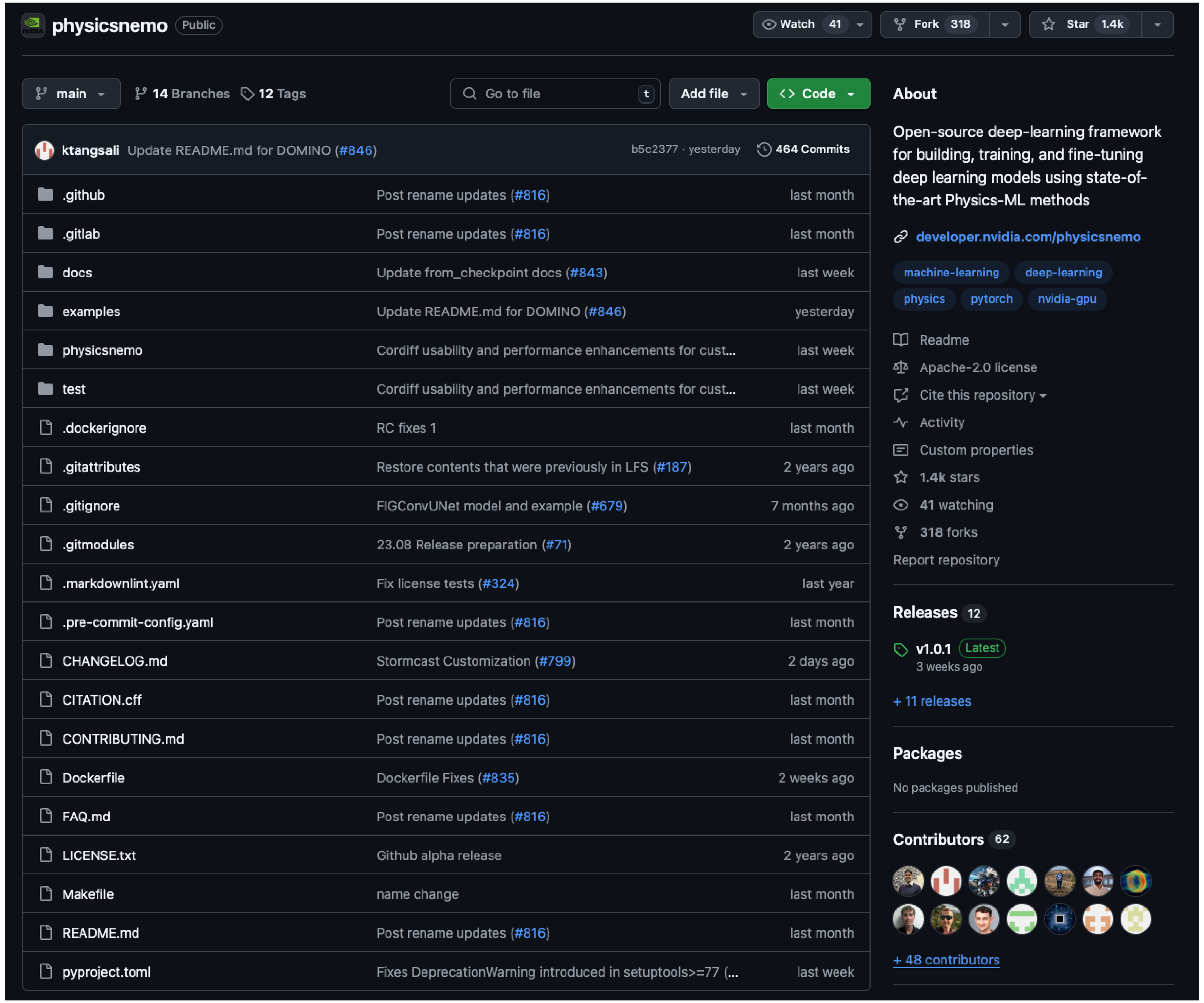
- Geometry Module
Integrated, parameterized geometry module with point cloud/SDF
- Symbolic PDE Loss Construction
Automated PDE loss construction using SymPy API
- Automated Optimized Gradient Calculations
Automatic gradient calculations for physics-informed learning with optimizations such as FuncTorch, AMP16, mesh free derivatives etc.
- Convergence and Stabilization Methods
Mass balance control planes, loss balancing schemes, AdaHessian support, learning rate annealing, etc.
- Exact Boundary Enforcement
Exact enforcement of continuity or boundary conditions
- Variational Learning
Solving PDE systems using variational formulations

PhysicsNeMo

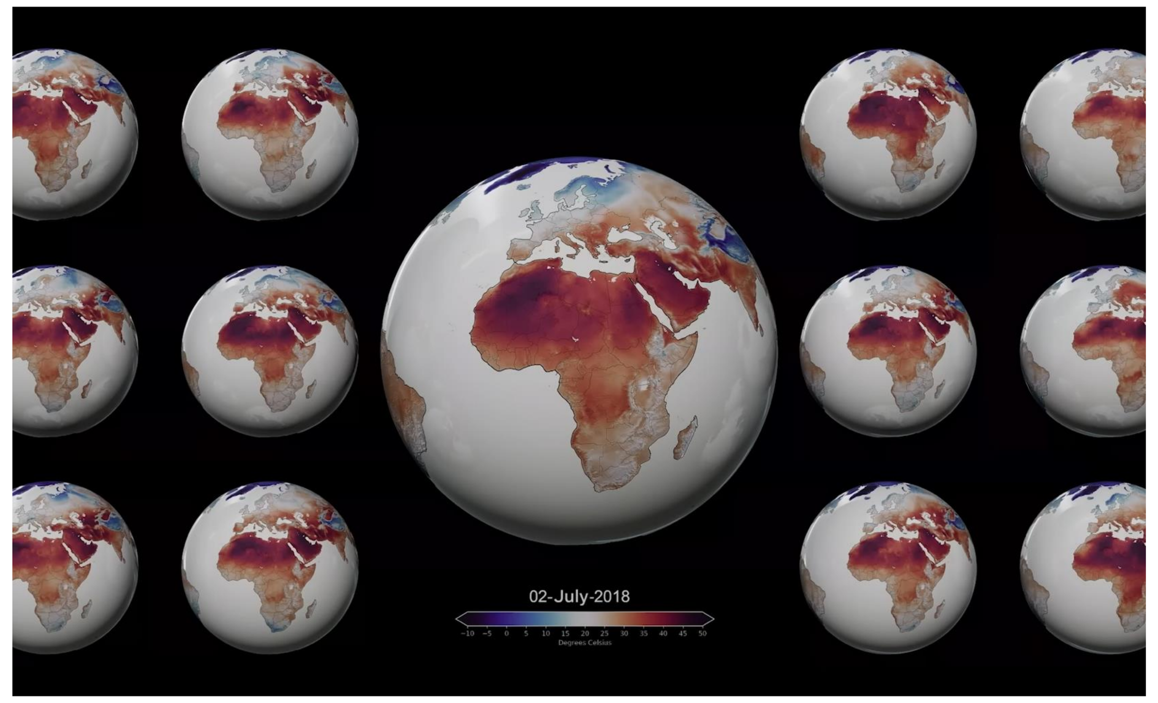
Open-Source on GitHub

- NVIDIA PhysicsNeMo is an **open-source** framework for building, training and fine-tuning Physics-ML models
- Available open-source on GitHub: https://github.com/NVIDIA/PhysicsNeMo





Physics-ML Success stories - PhysicsNeMo Case-Studies

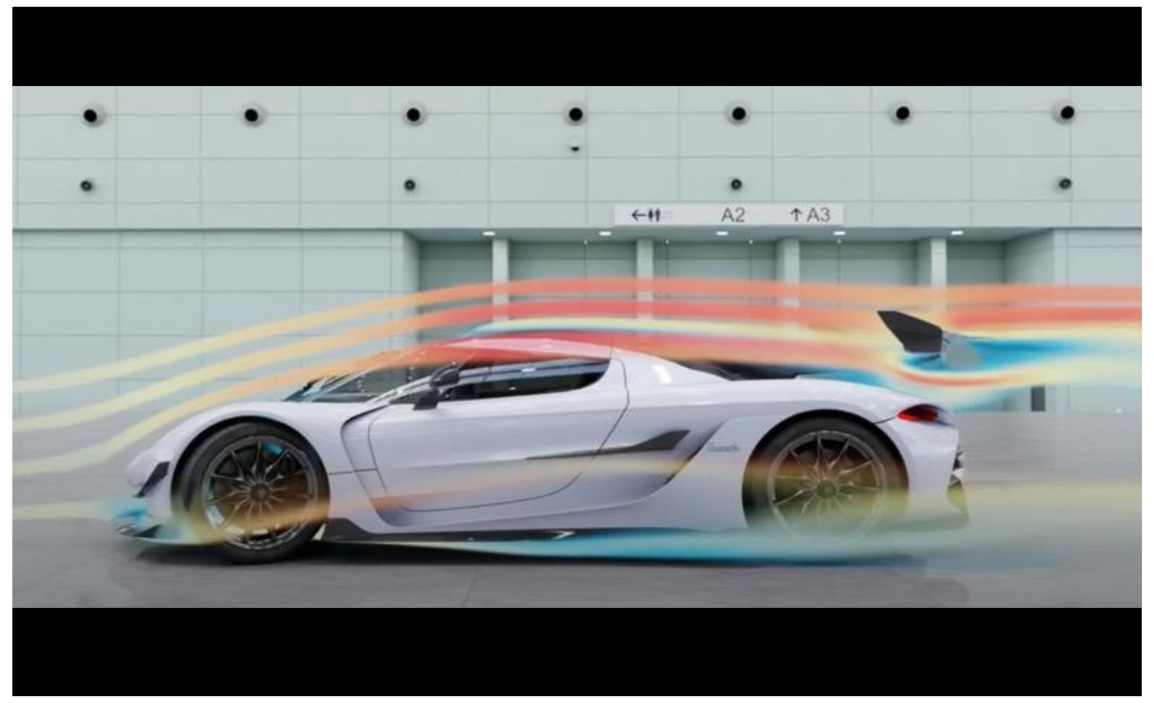


Weather modeling

Demo: Link 1, Link 2, Link 3

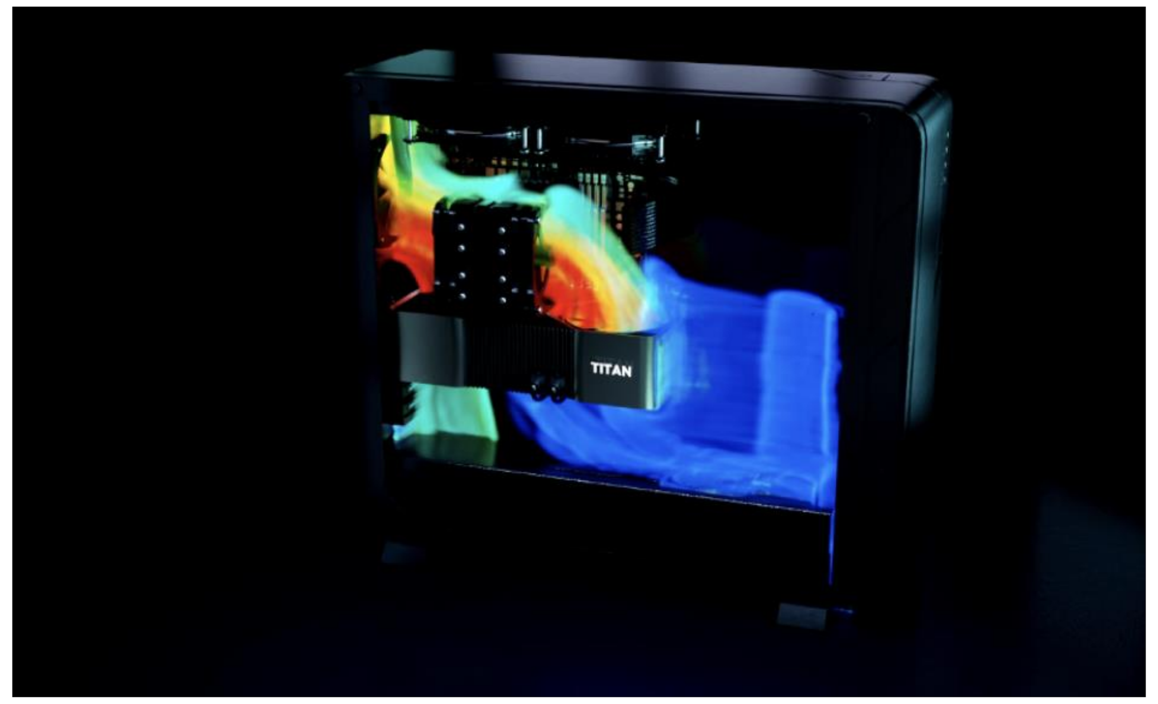
Wind farm Super Resolution

Demo: Link, Blog: Link



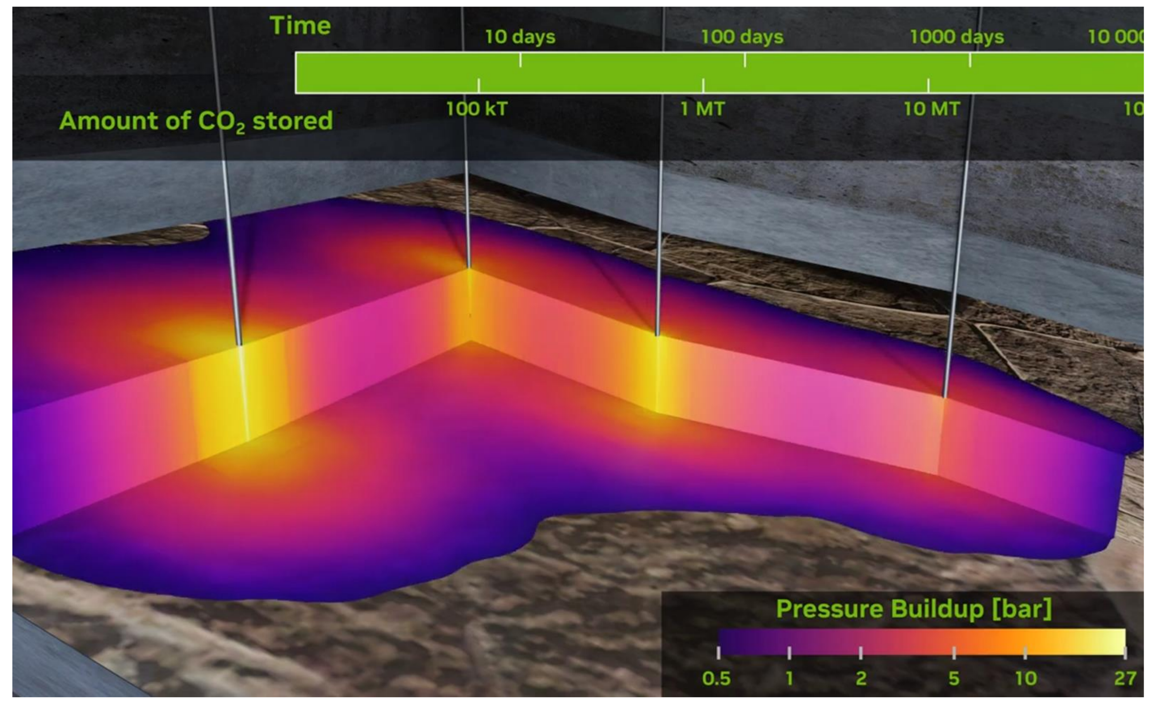
Automotive CFD

Omniverse Blueprint, NIM



RTX 4090 heat sink design

Demo: Link

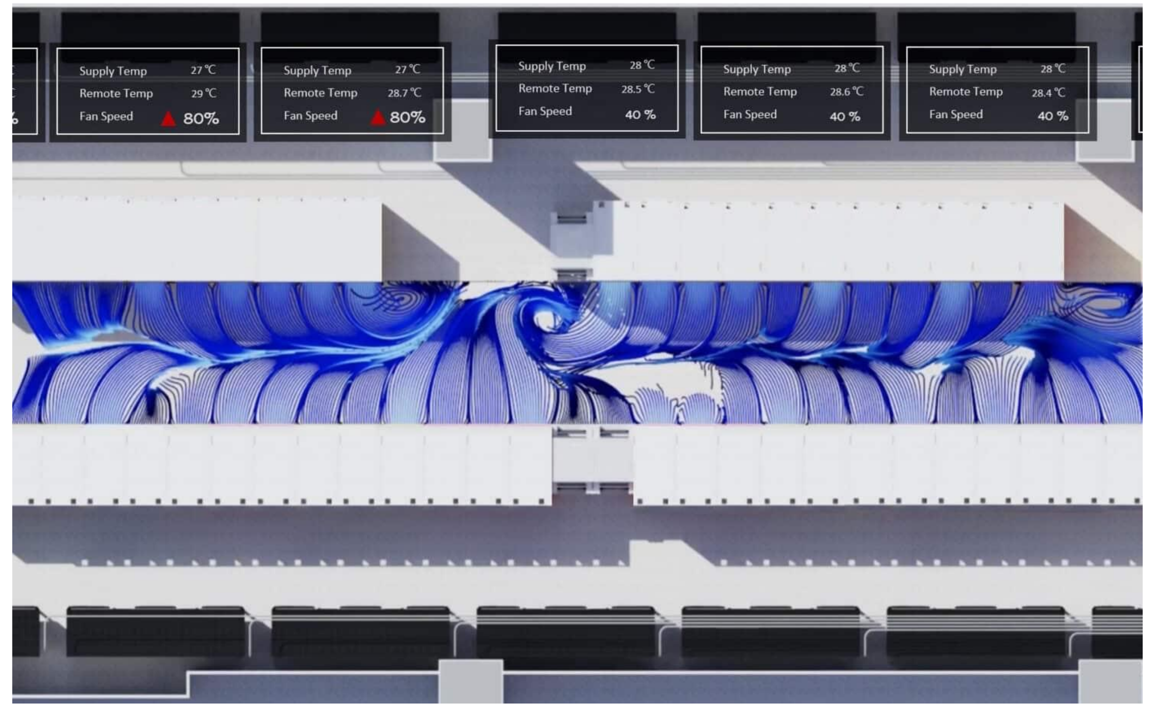


Carbon capture and storage

Demo: Link, Blog: Link

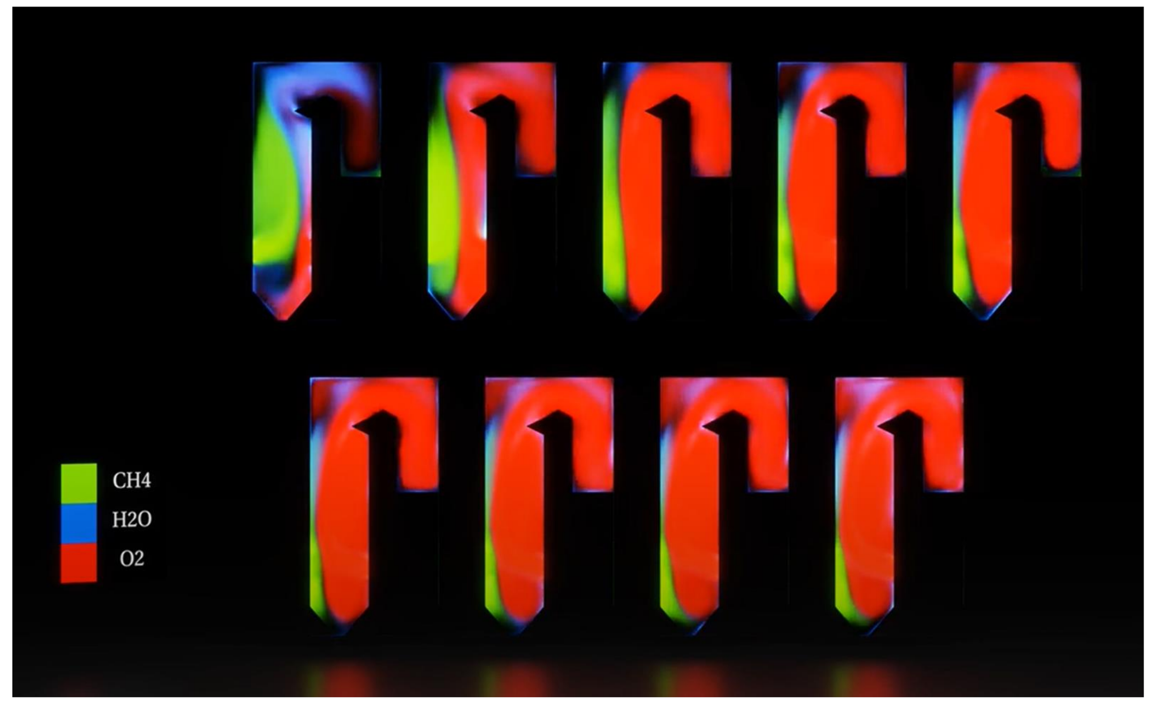
HRSG Digital Twin

Demo: Link, GTC Session: Link



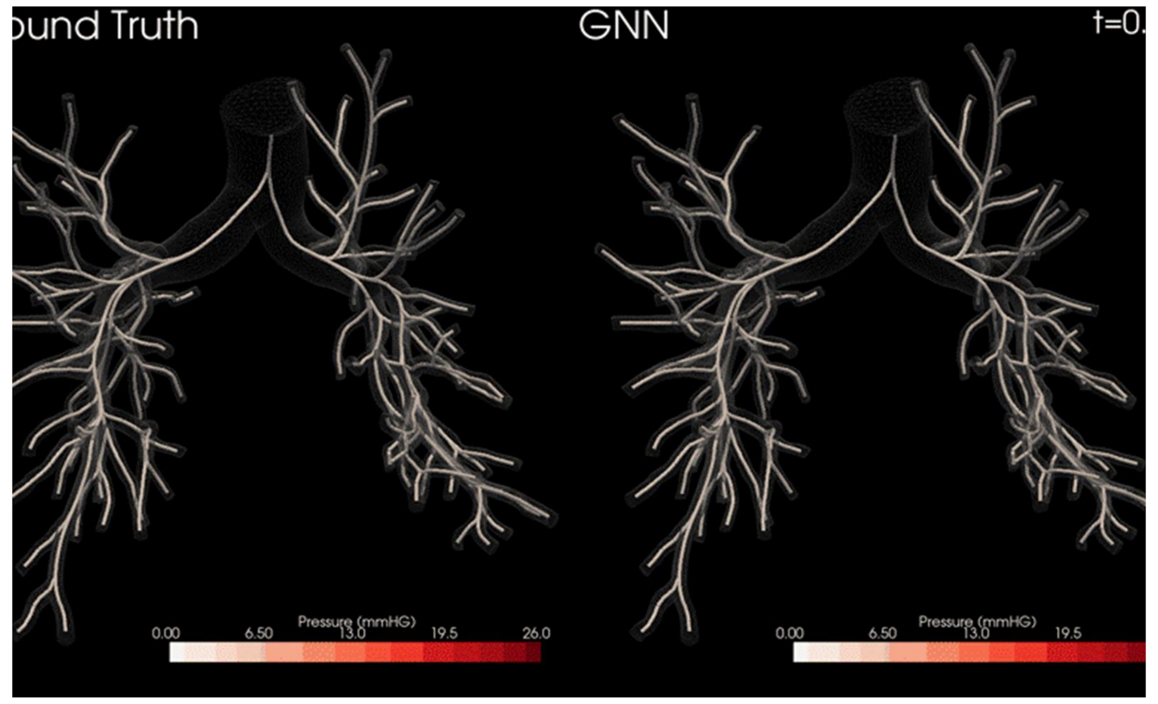
Data Center Digital Twin

Blog: Link, GTC Session: Link



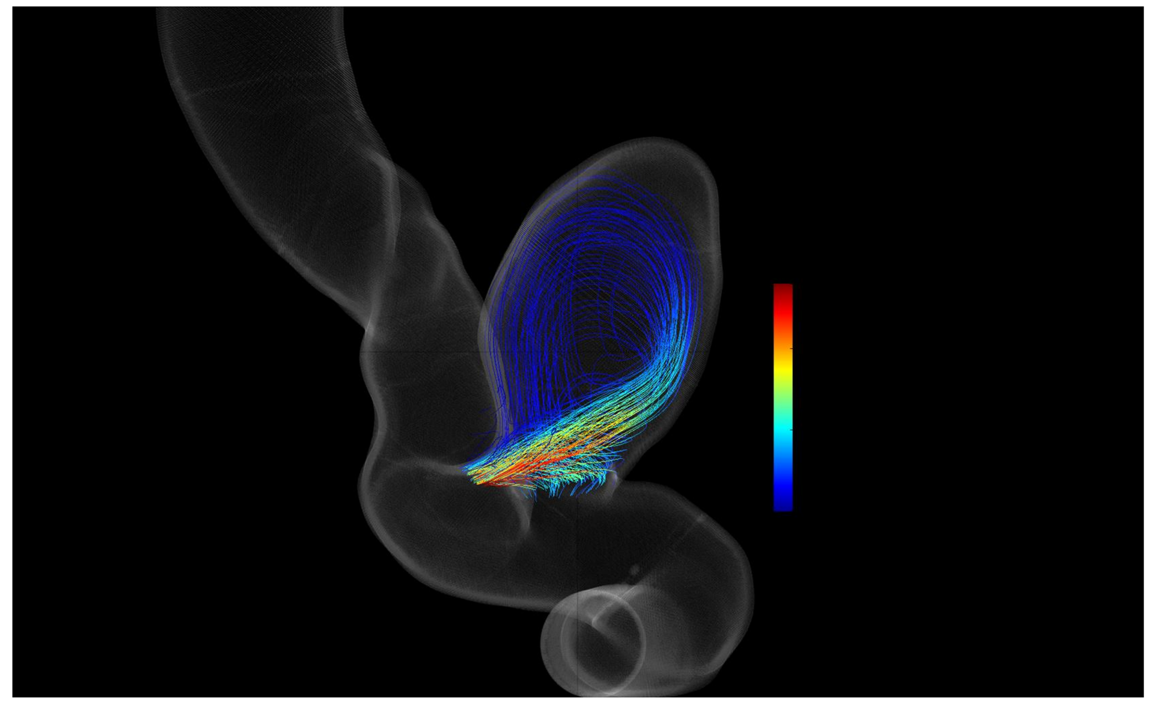
Thermal Boiler Digital Twin

Blog: Link, GTC Session: Link



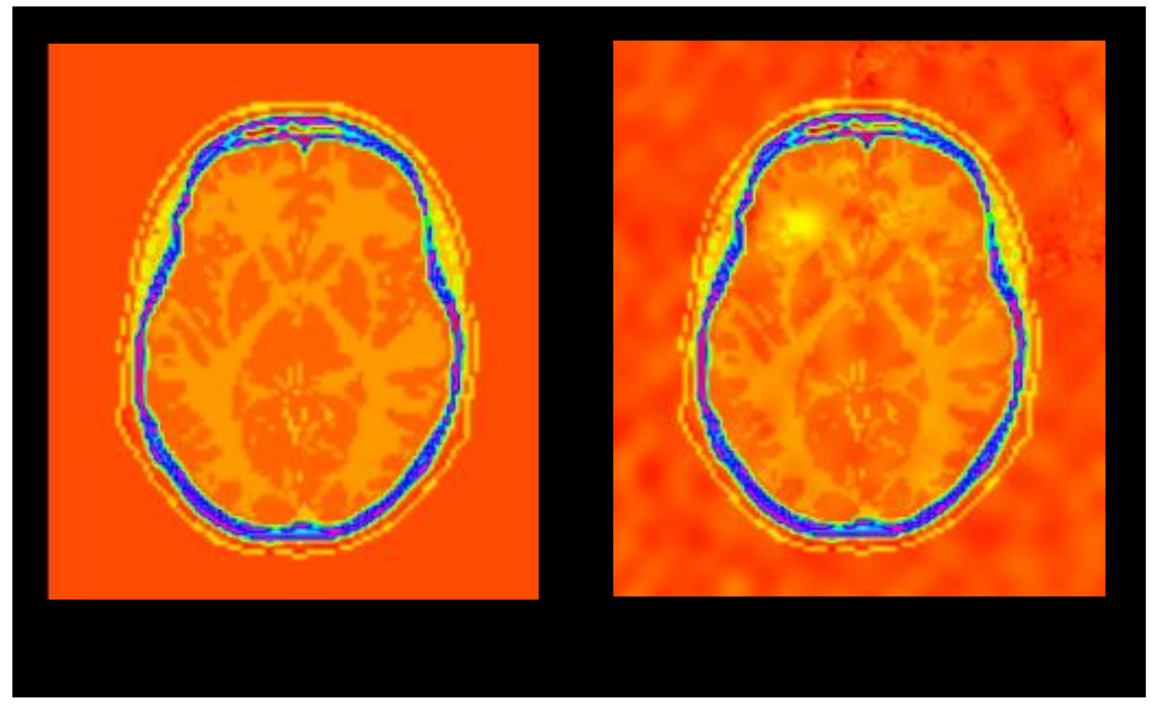
Cardiovascular Simulation

Blog: Link



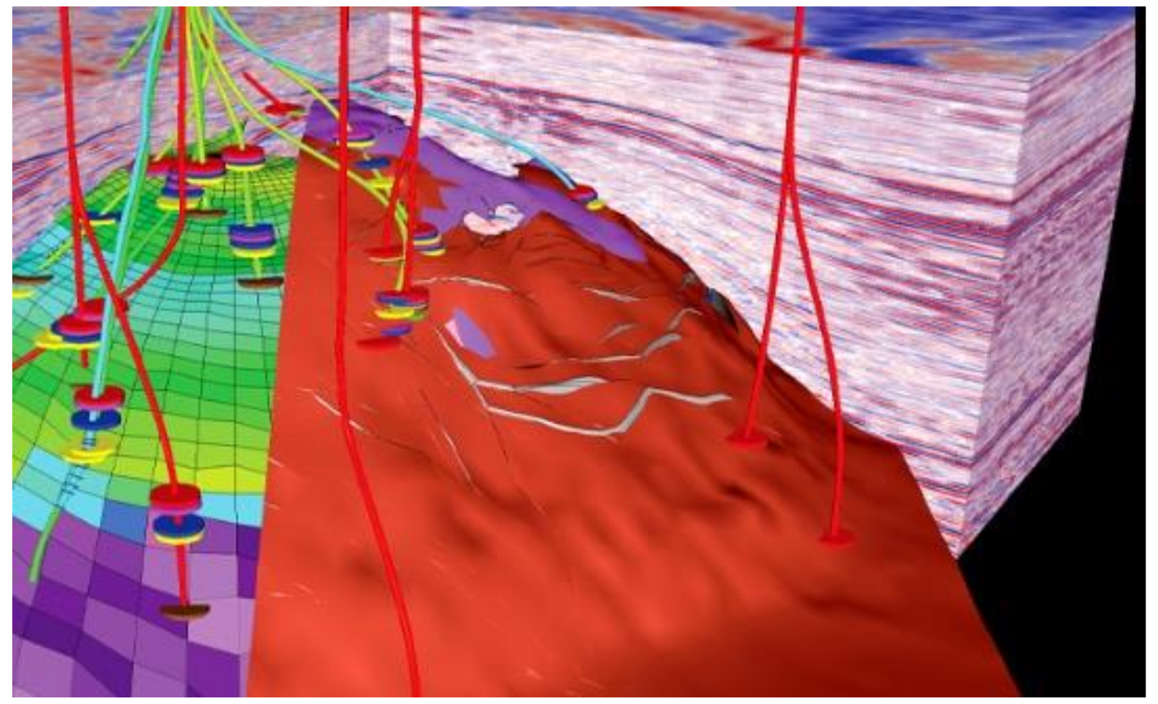
Brain Aneurysm Simulation

Demo: Link



Brain Anomaly Detection

Resource: Link



Sub surface simulations

Resource: Link

OpenACC OPEN HACKATHONS

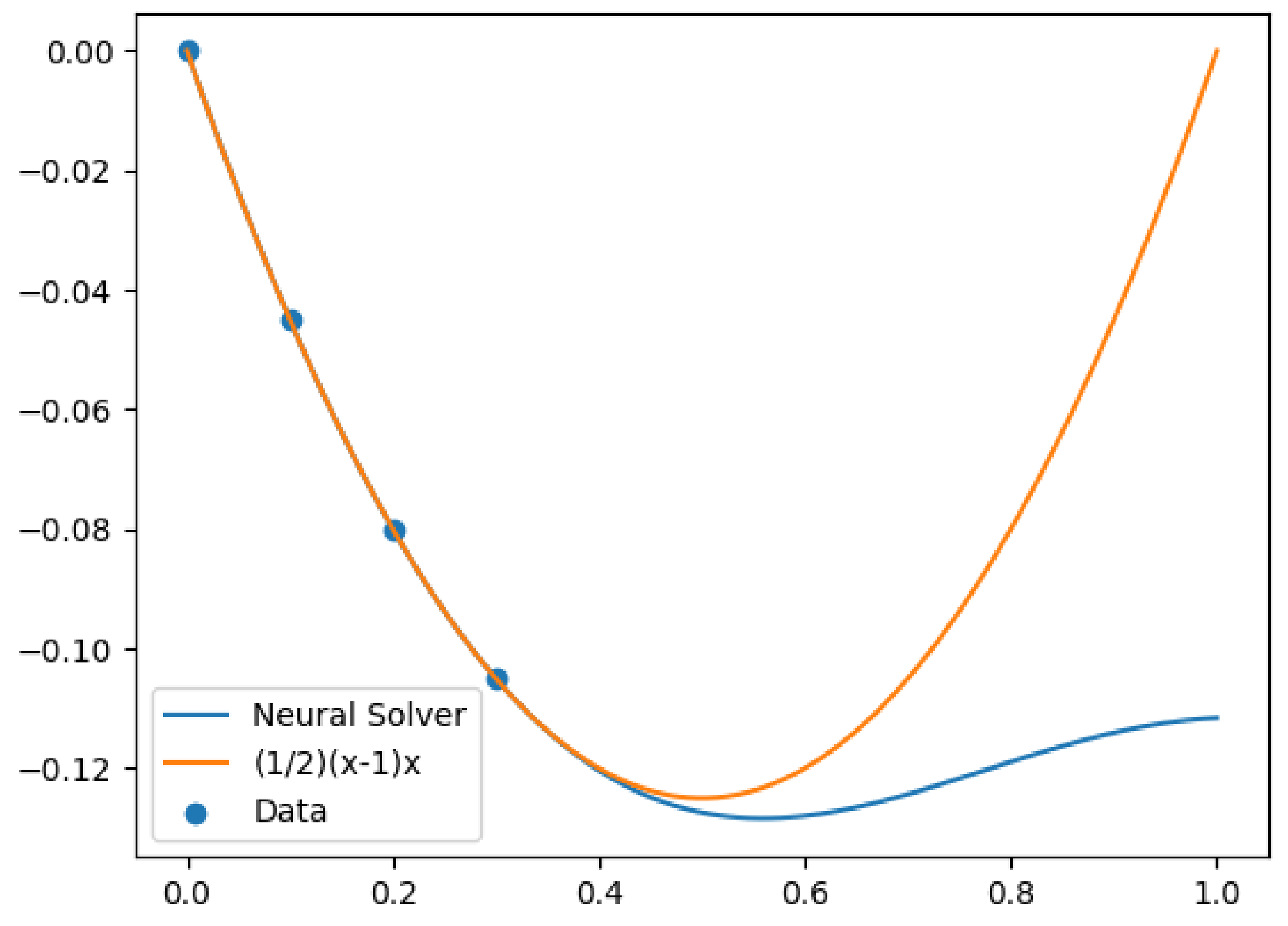
Note: This session uses PhysicsNeMo 25.03 Release

Adding Physics laws as soft constraints

Data Only

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

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

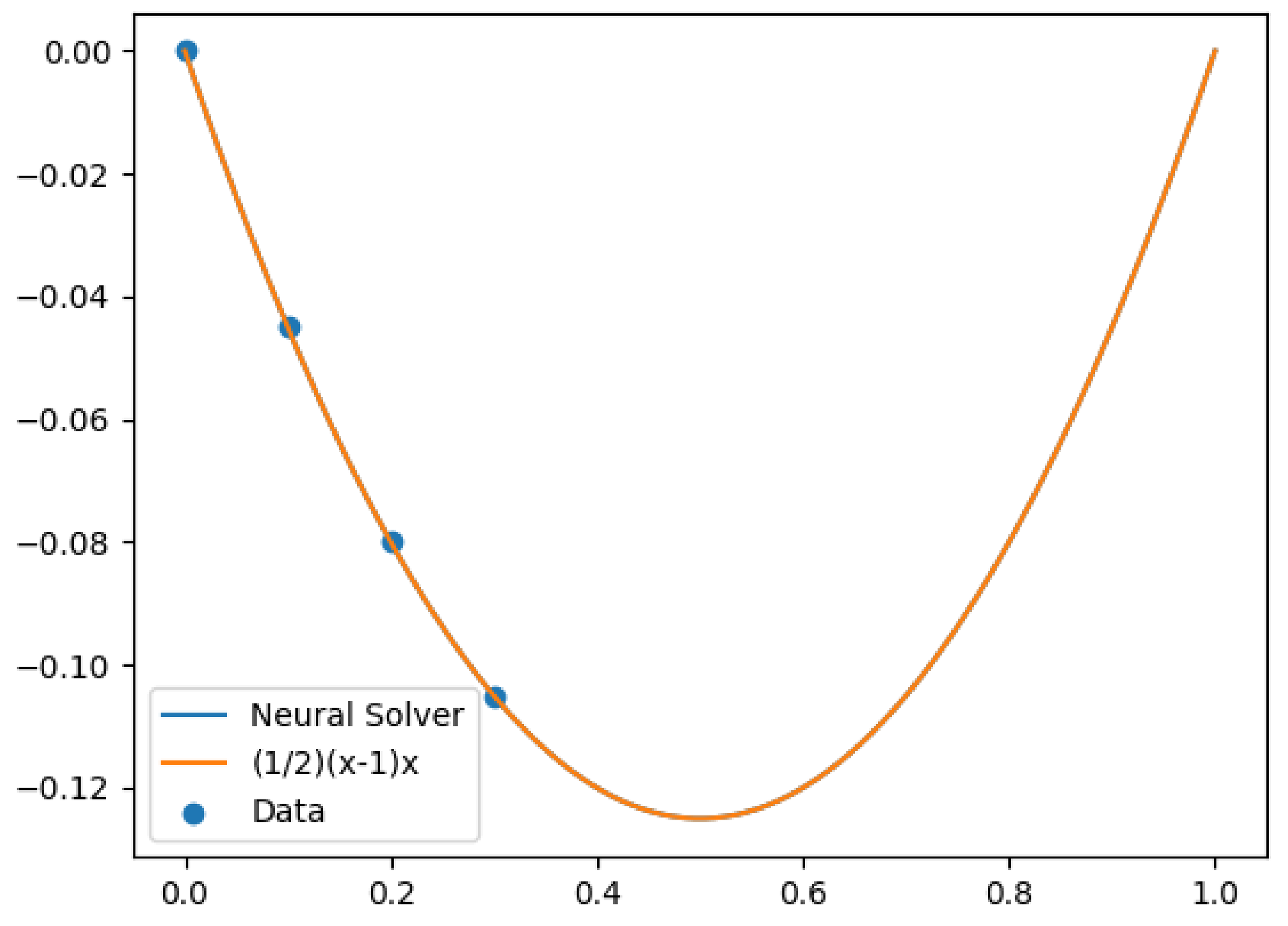


Data + Physics

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

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

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

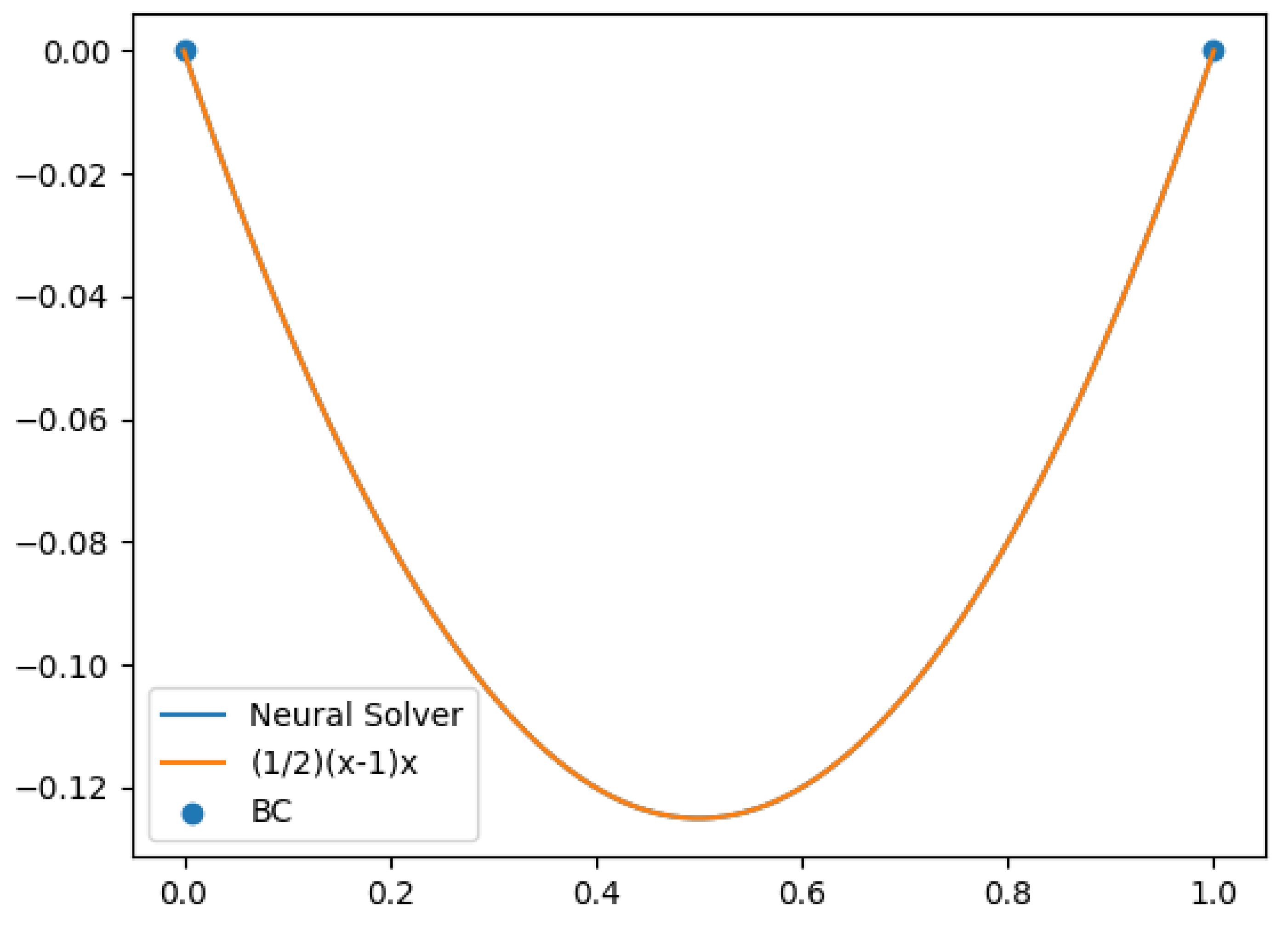


Physics only

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

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

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



Note, this is only one of the many possible ways to incorporate physics knowledge in the model training!

OpenACC OPEN HACKATHONS

Computing Physics loss as soft constraints

Train a neural network using only physical constraints

- Consider an example problem:

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

- To solve the PDE using only the equation and BCs, 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

Computing Physics loss as soft constraints

Train a neural network using only physical constraints: 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, compute the integrals using Monte-Carlo integration technique

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

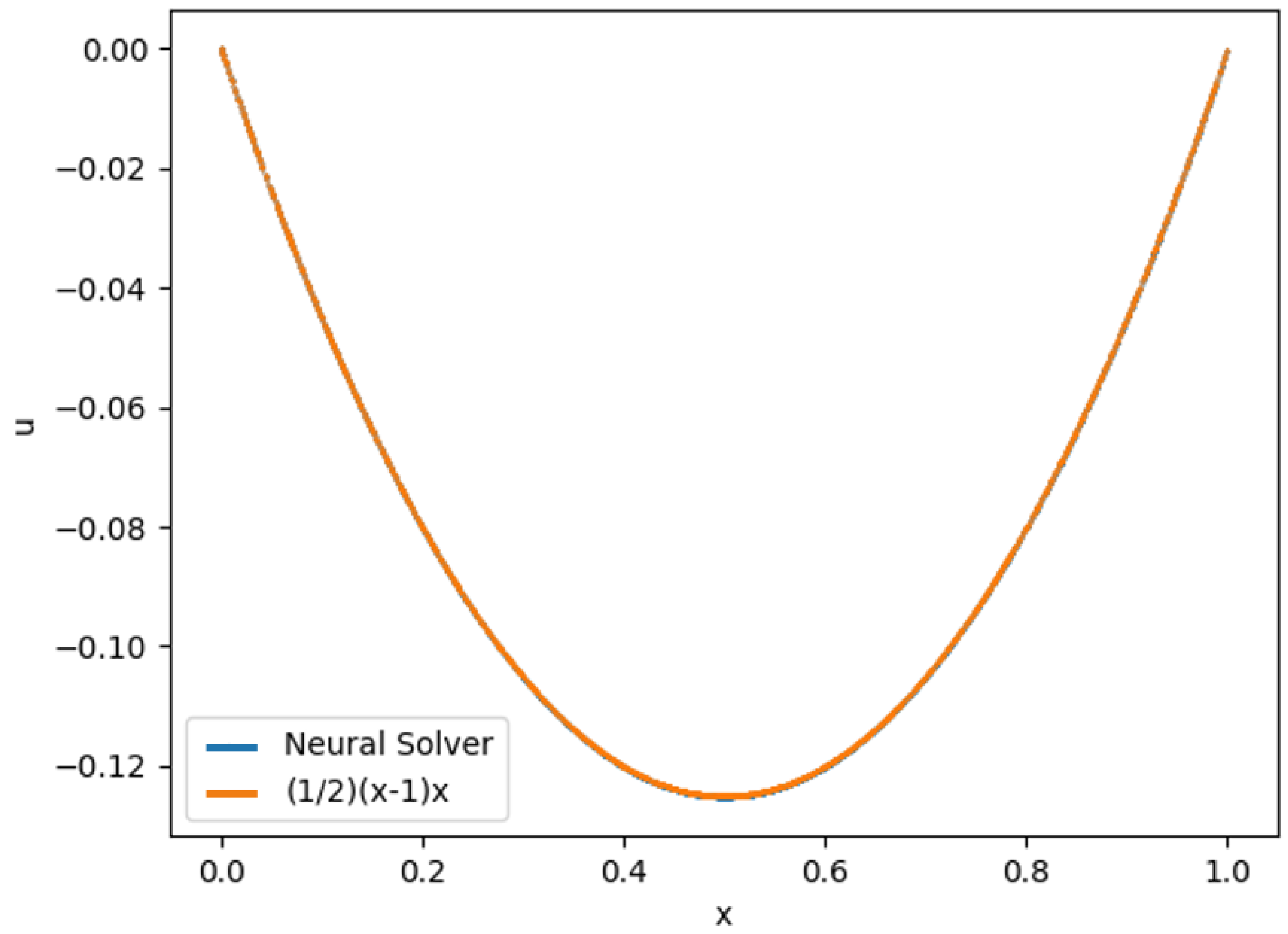
$$L_{Residual} = \int_0^1 \left(\left(\frac{\delta^2 u_{net}}{\delta x^2}(x) - f(x) \right) - 0 \right)^2 dx \approx \left(\int_0^1 dx \right) \frac{1}{N} \sum_{i=0}^{N} \left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i) \right)^2$$

- 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

Computing Physics loss as soft constraints

Train a neural network using only physical constraints: Results

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



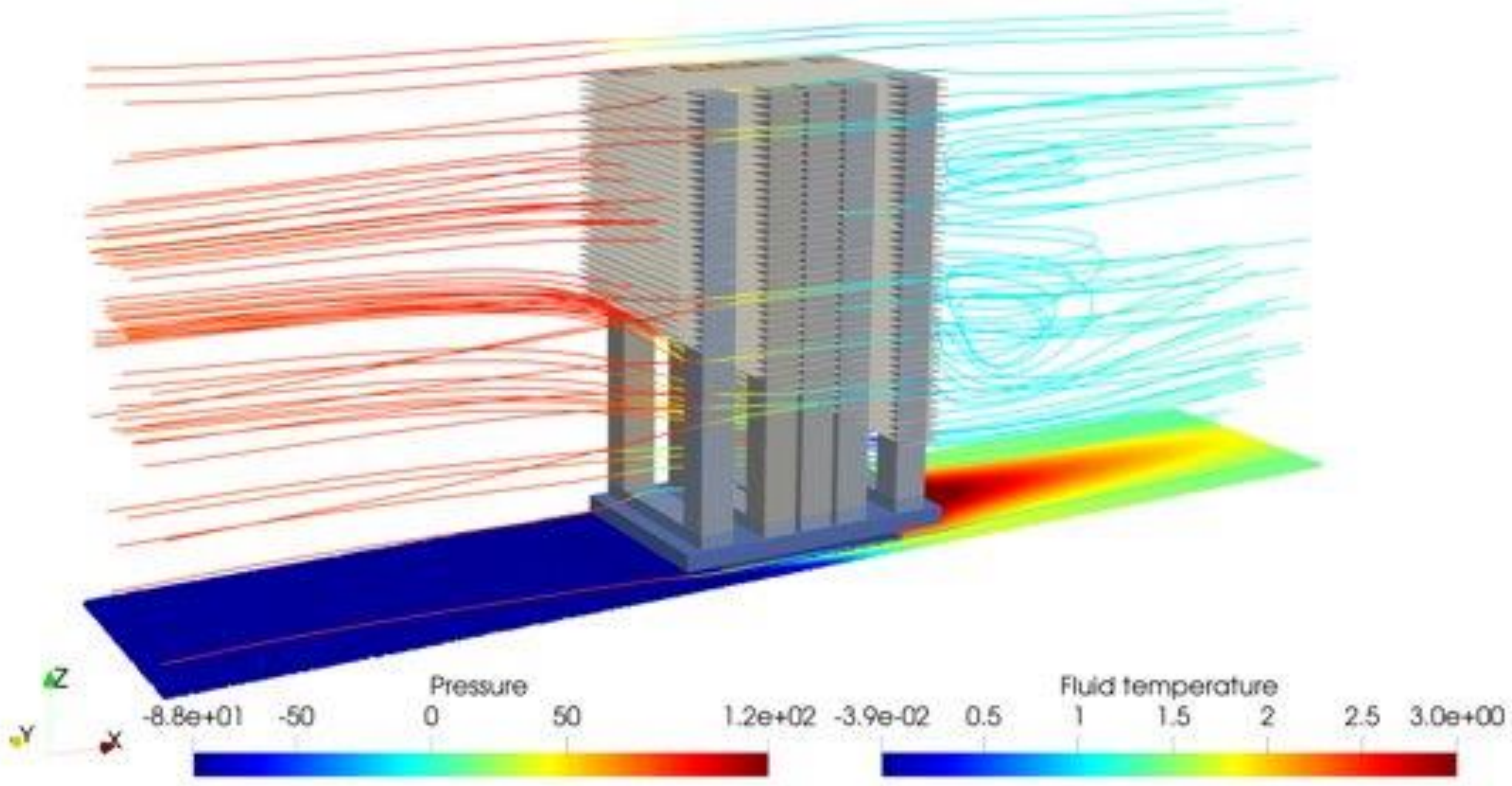
Comparison of the solution predicted by Neural Network with the analytical solution

PhysicsNeMo Sym

Framework for AI surrogates using Physics-Based symbolic loss functions

- High-level abstract training framework for training physics-constrained models
- Flexible multi-constraint training workflow with many physics-driven training enhancements
- Symbolic paradigm (keys) which enables gradient calculation automation

KEY FEATURES:
Abstraction: High level API for domain experts
Automation: Automated loss and gradient calculations for PINNs training using SymPy
Parallelism: Automated DDP and model parallel setup
Optimization: Similar optimizations as PhysicsNeMo-Launch with some additions (PINNs AMP)
Monitoring / Logging: Tensorboard
Hydra: Configurable through Hydra
Example Documentation: Comprehensive set of examples for different physical systems

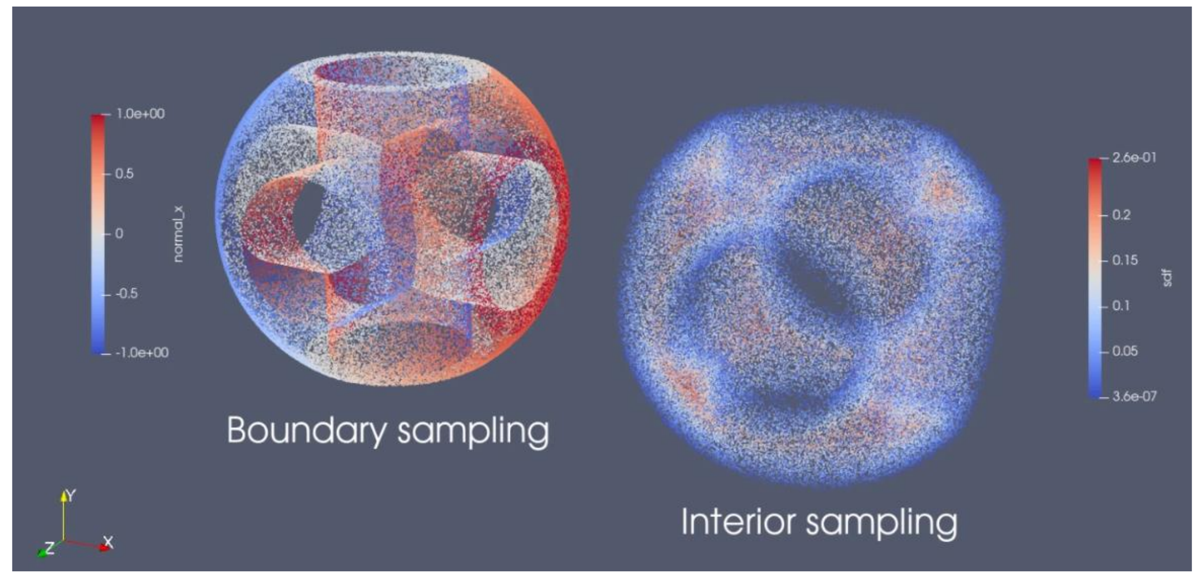


OpenACC OPEN HACKATHONS

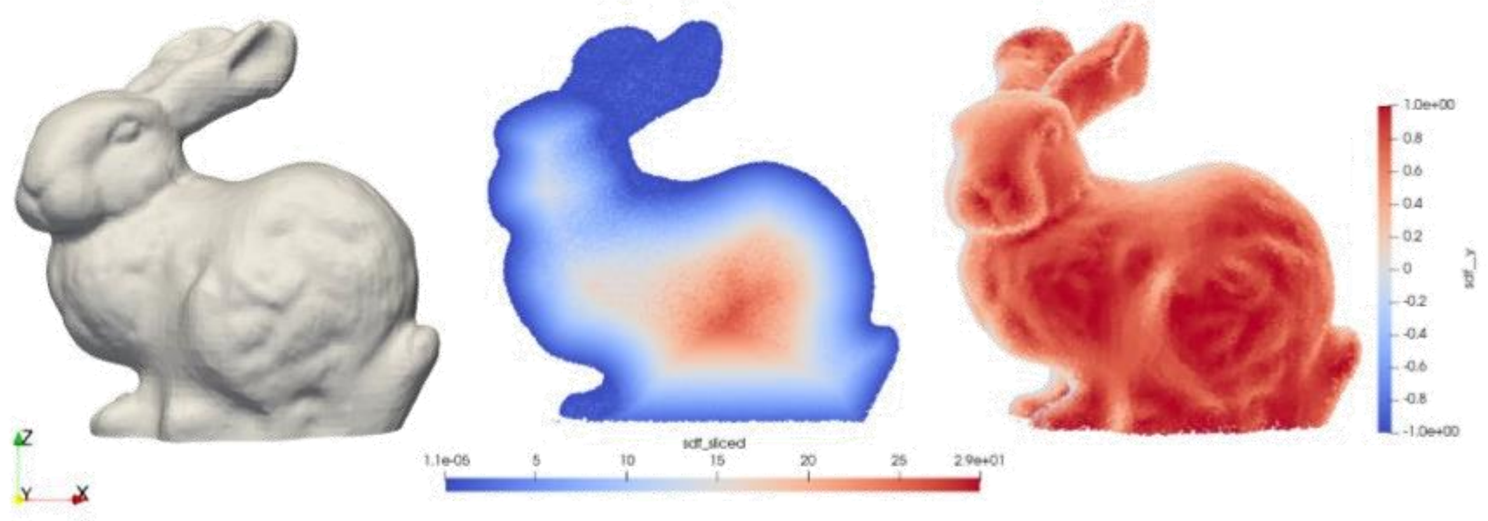
Physics guided training workflows

Leveraging abstracted utils from PhysicsNeMo Sym

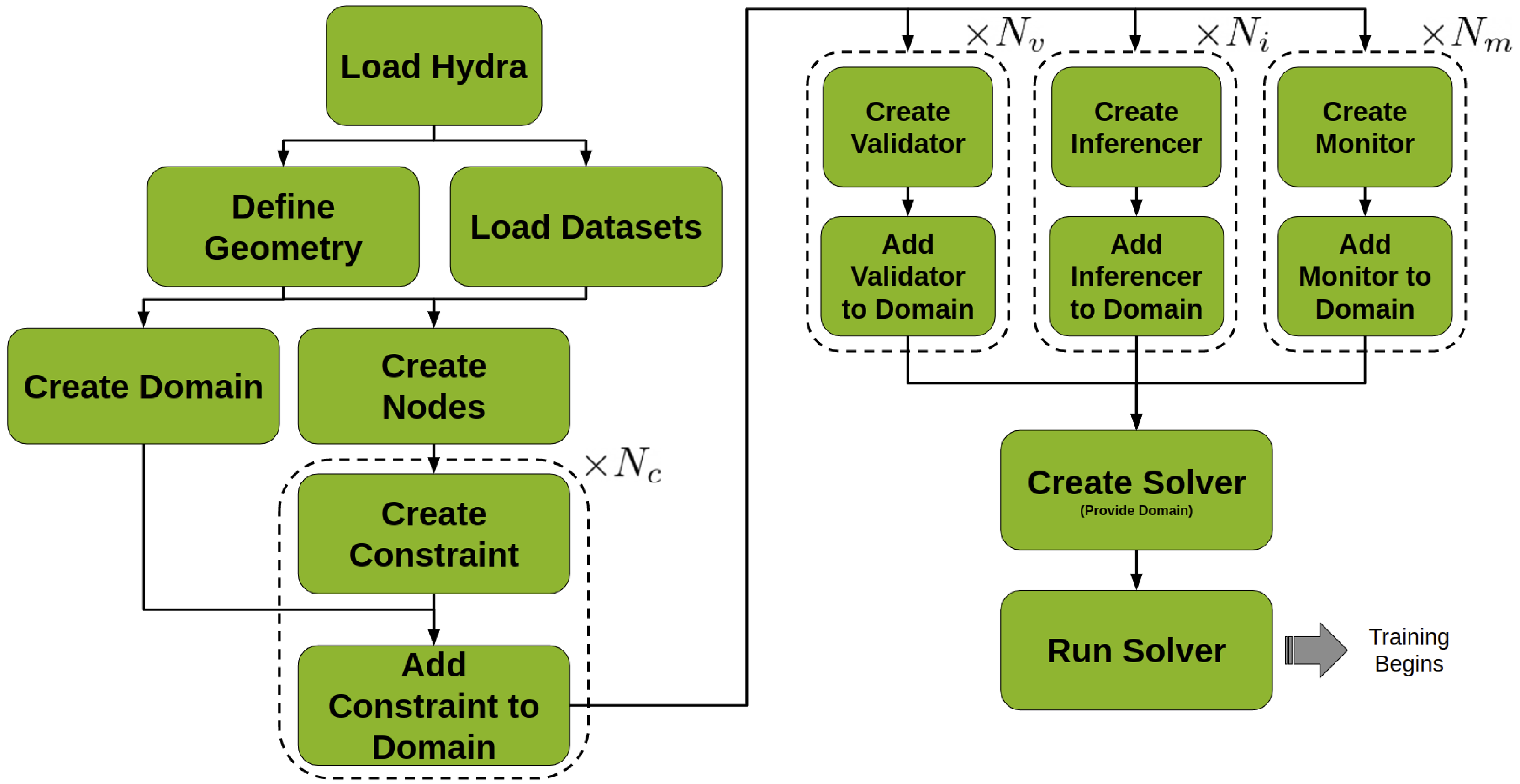
- Key questions:
  - How to determine domain of interest?
    - Full domain
    - Boundaries, …
  - How to sample the domain?
    - Collocation points
    - Test functions
    - Discretization, …
  - How to specify the constraints/losses information?
    - Control Volume Formulation
    - DifferentiVariational Formulation, …
    - al Formulation
- PhysicsNeMo Sym has utilities to simplify such problem setups. Utils can be used **standalone** or in **abstracted training definition** framework



Constructive Solid Geometry



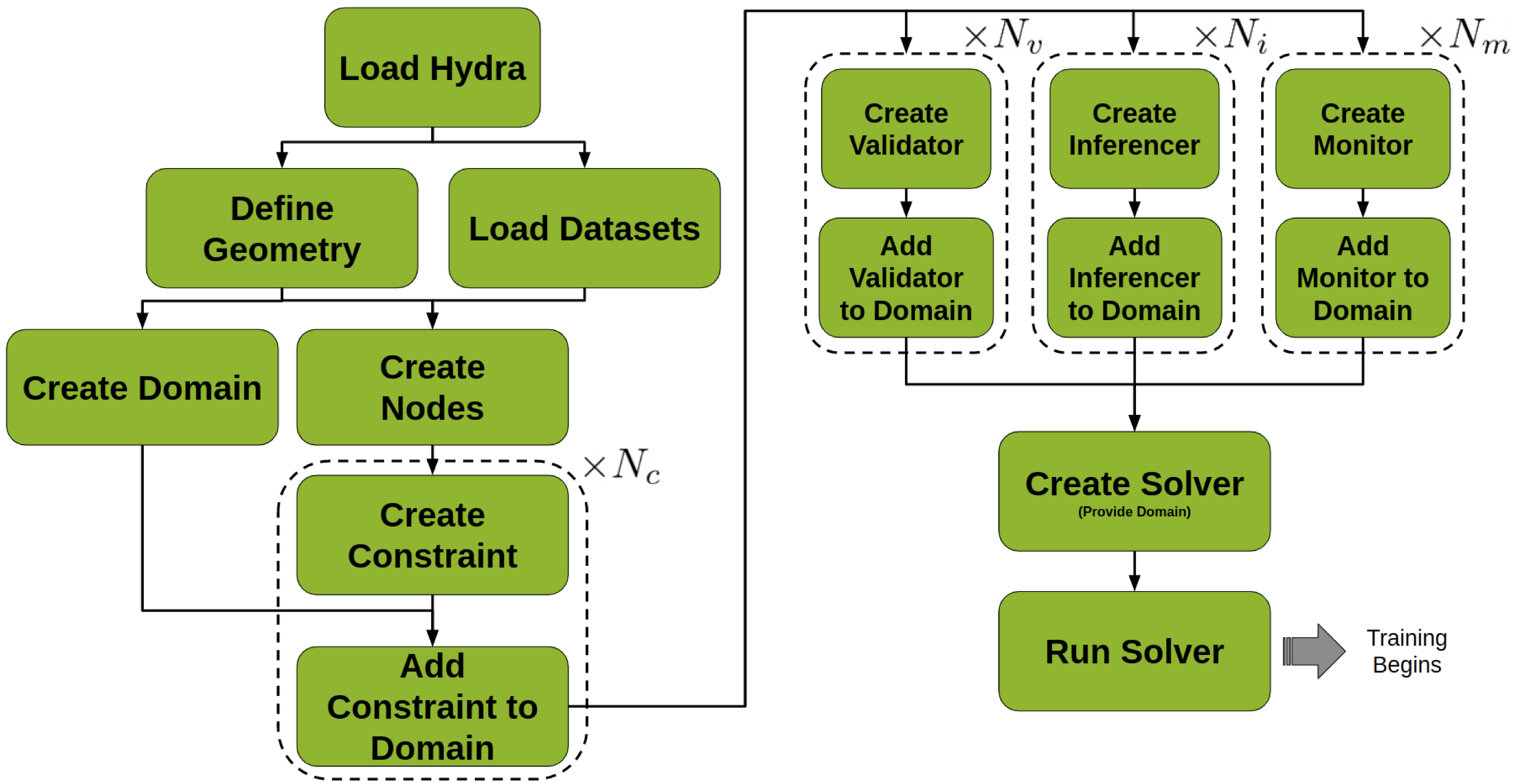
Tessellated Geometry (STL)



PhysicsNeMo Sym's abstracted Training workflow

PhysicsNeMo Sym: Anatomy of a project

Overview



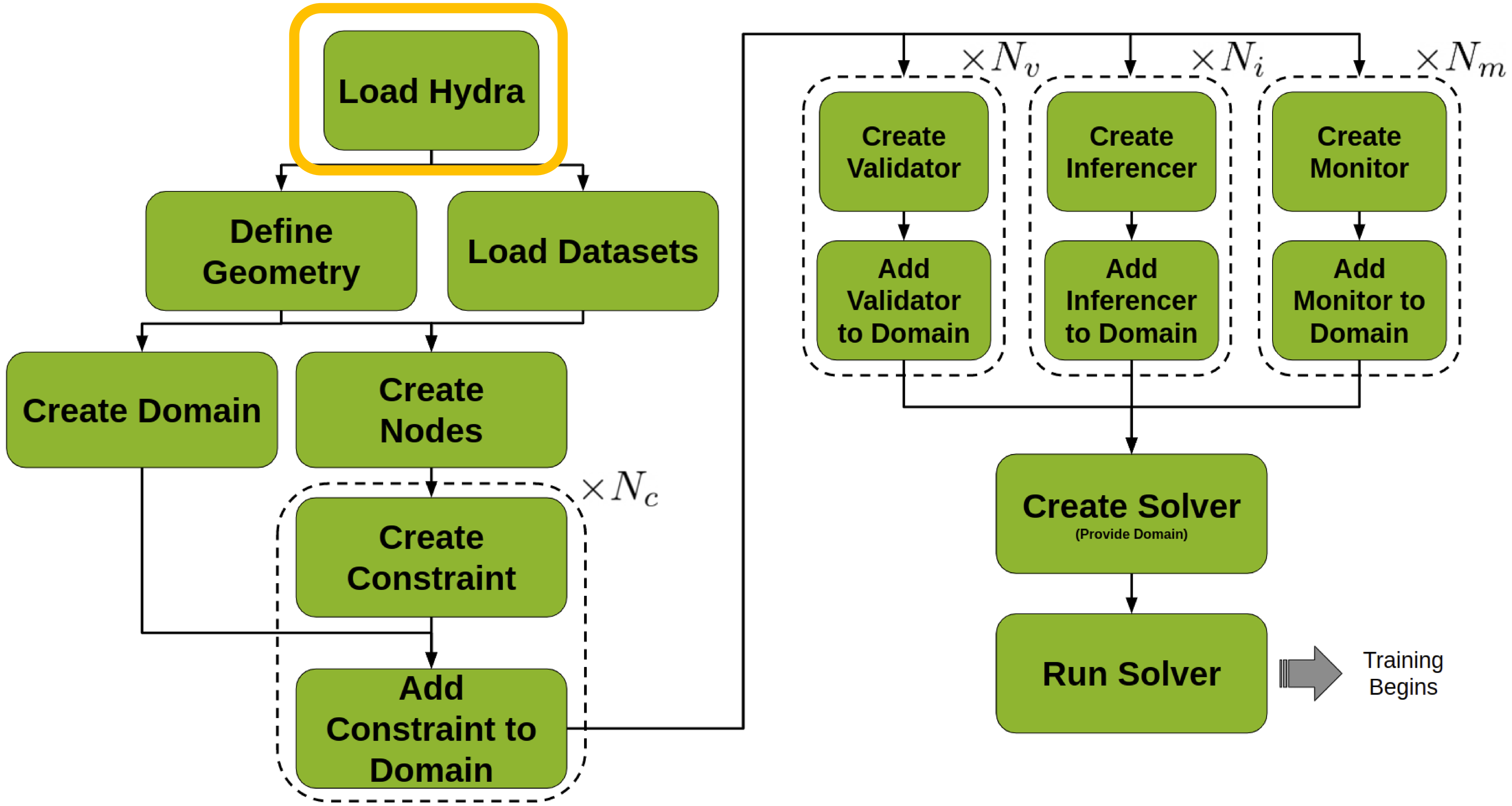
PhysicsNeMo Sym works by:

- 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

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

Physics guided training workflows

PhysicsNeMo Sym approach: Load Hydra

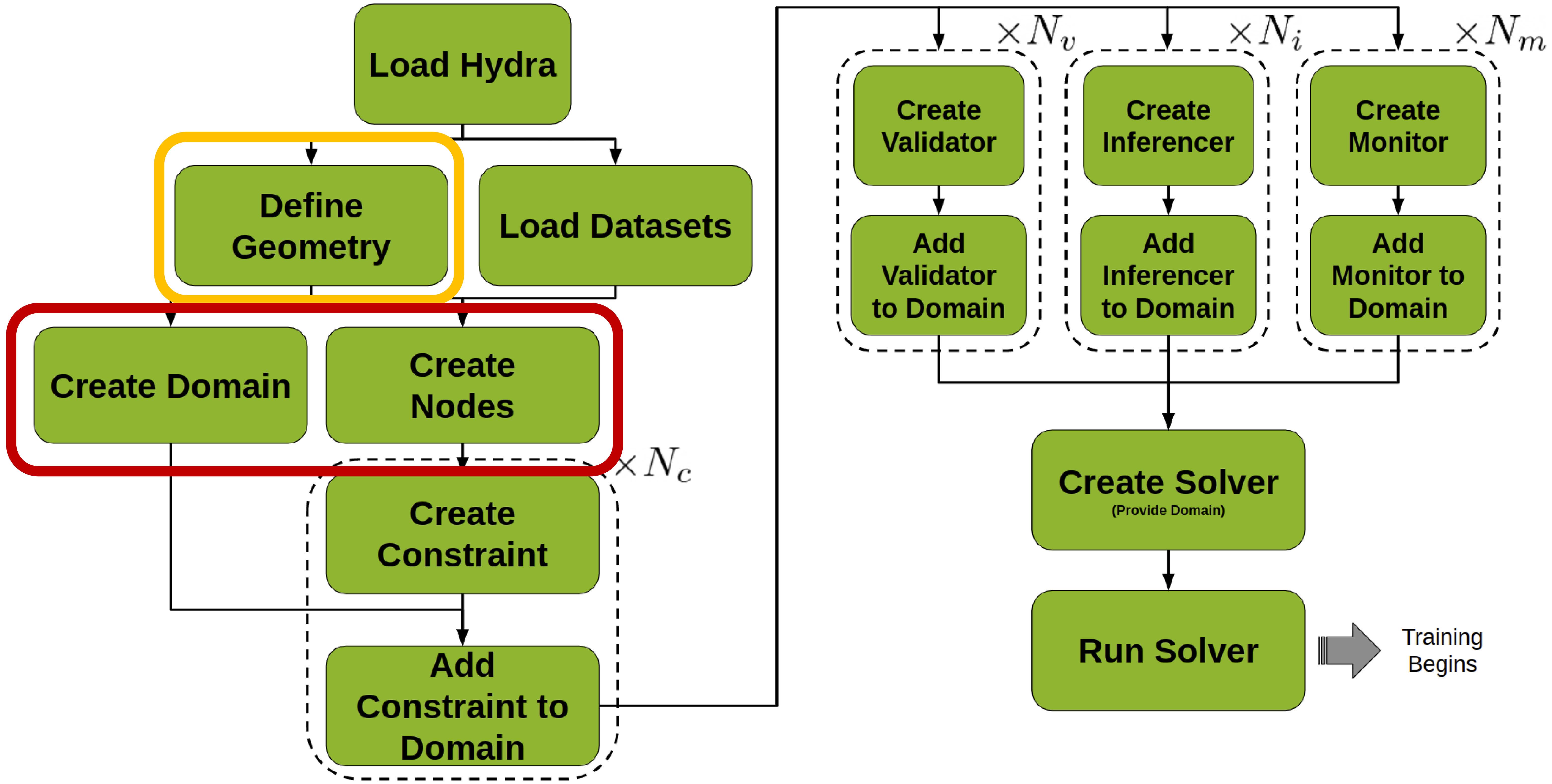



```yaml
defaults :
- Physicsnemo_default
- scheduler: tf_exponential_lr
- optimizer: adam
- loss: sum
- _self_
scheduler:
decay_rate: 0.95
decay_steps: 200

save_filetypes : "vtk,npz"

training:
rec_results_freq : 1000
rec_constraint_freq: 1000
max_steps : 5000
```

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

Physics guided training workflows

PhysicsNeMo Sym approach: Create geometry, domain, and nodes




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

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

```python
    # 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()
```

Use the model architectures from PhysicsNeMo-Sym that are designed to work with Auto-grad out-of-the-box

PhysicsNeMo:

- physicsnemo.sym.geometry contains implementations of 1D, 2D and 3D primitives that can be assembled to compose complex geometries
- physicsnemo.sym.tessselation enables import of STL geometries
- Sample point cloud inside and on the surface of generated geometries

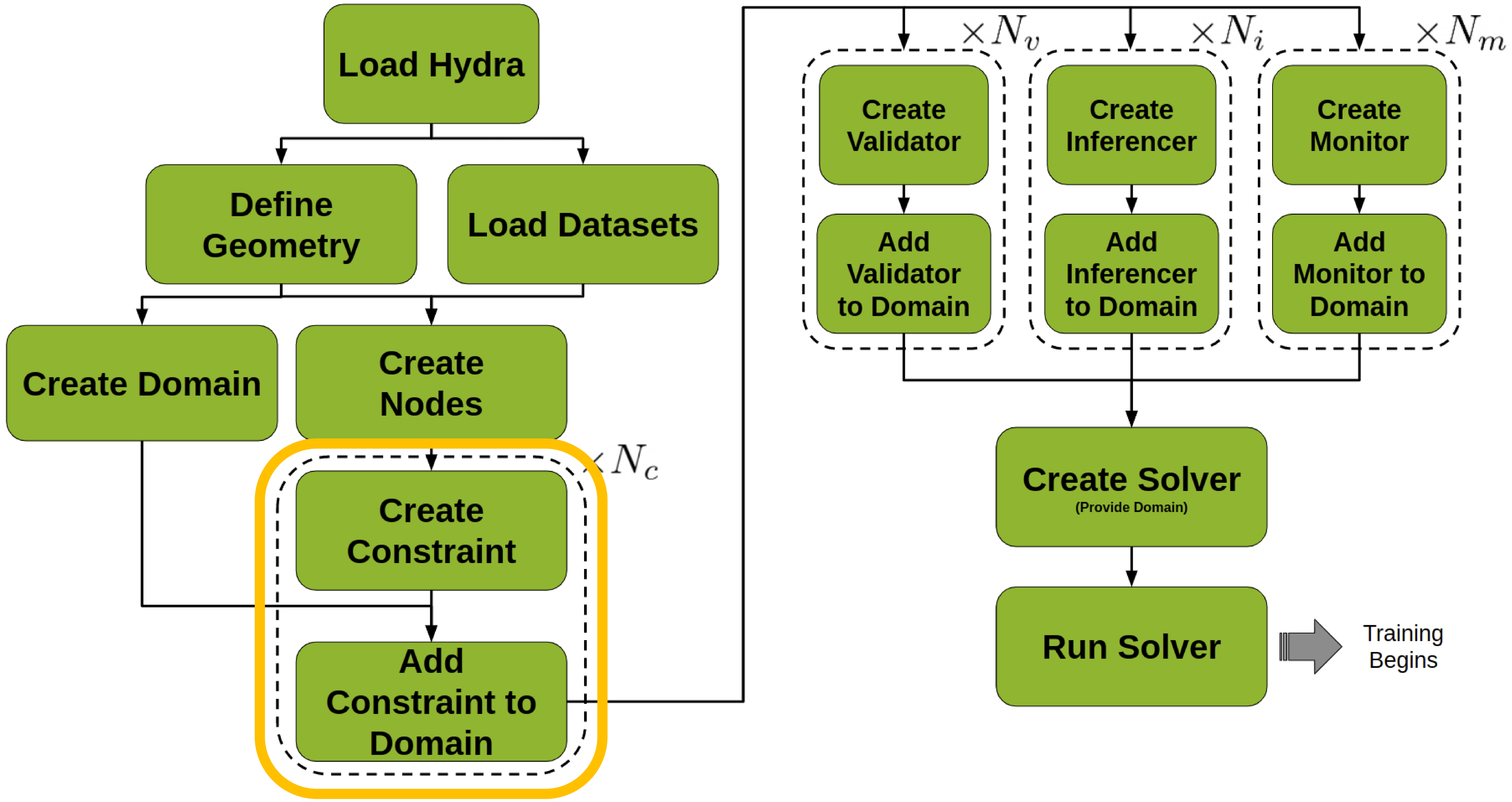
PDE and computational nodes:

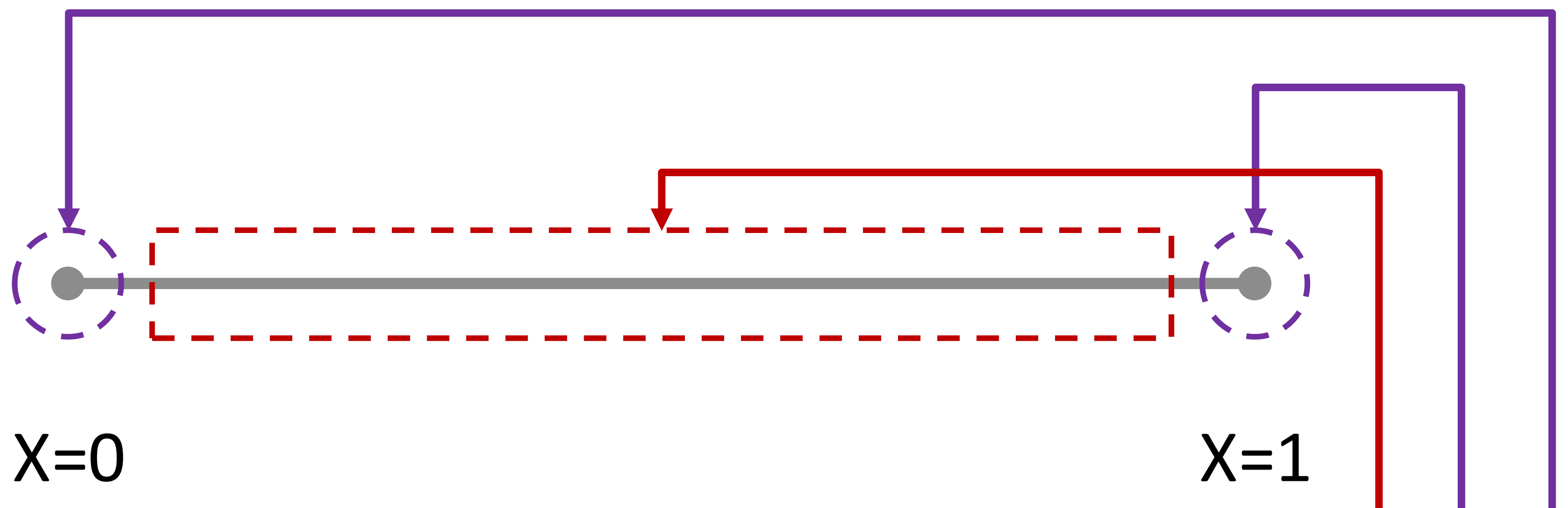
- Implementations of fundamental governing equations from domains like: Fluid Mechanics, Linear Elasticity, Electromagnetic, etc.

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

Physics guided training workflows

PhysicsNeMo Sym approach: Add constraints





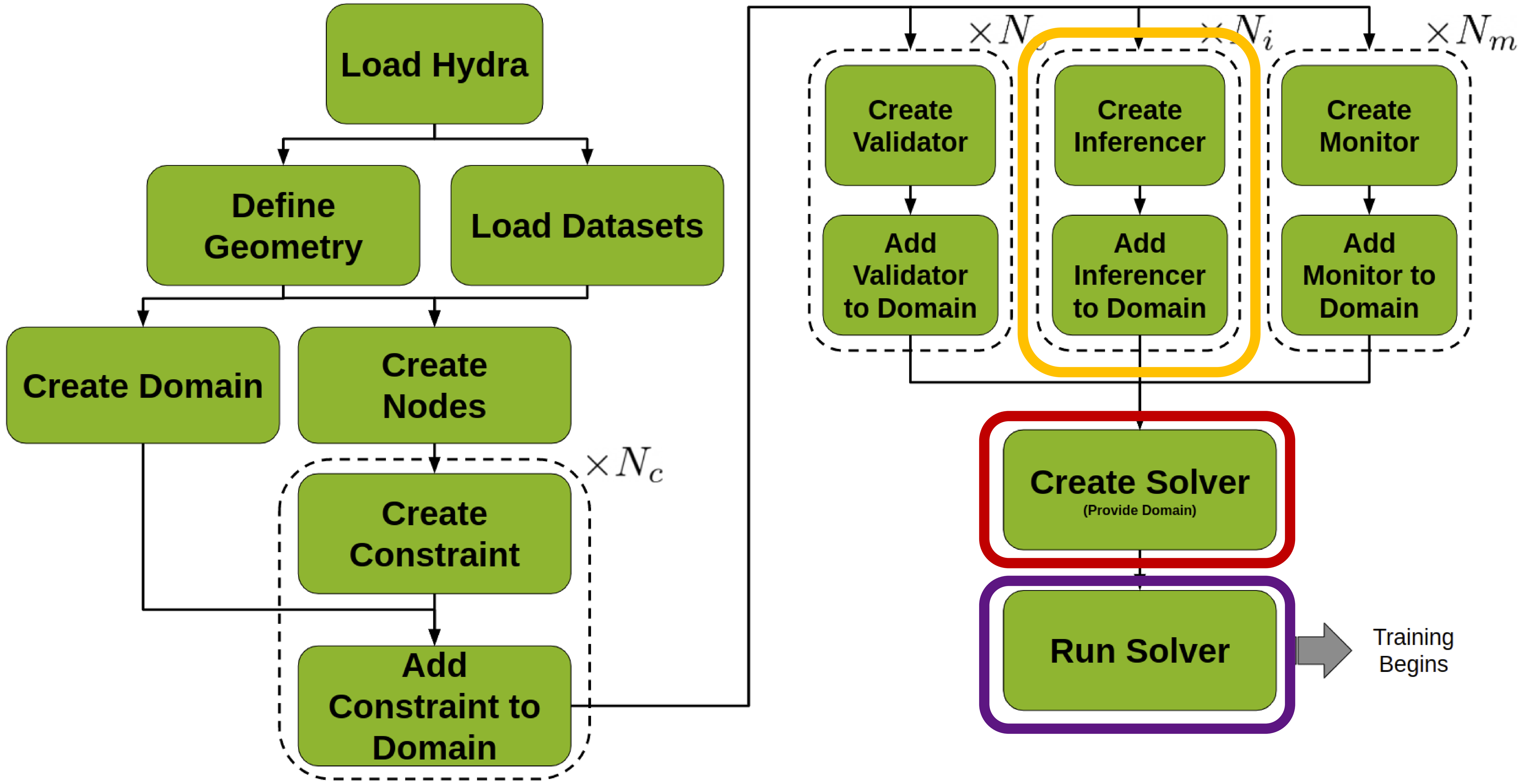

```
    # add constraints

    # 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")
```

Automatically select the boundary and interior points

Physics guided training workflows

PhysicsNeMo Sym approach: Add utils to visualize results, trainer loop

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




```
# add inferencer
inference = PointwiseInferencer(
nodes=nodes,
invar={"x": np.linspace(0, 1.0, 100).reshape(-1,1)},
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
```

Use the pre-defined optimized training loop

Abstracted and Optimized training loop:

- Solver and Trainer class abstract a lot of complexity of defining the Neural network training allowing users to focus on problem definition

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

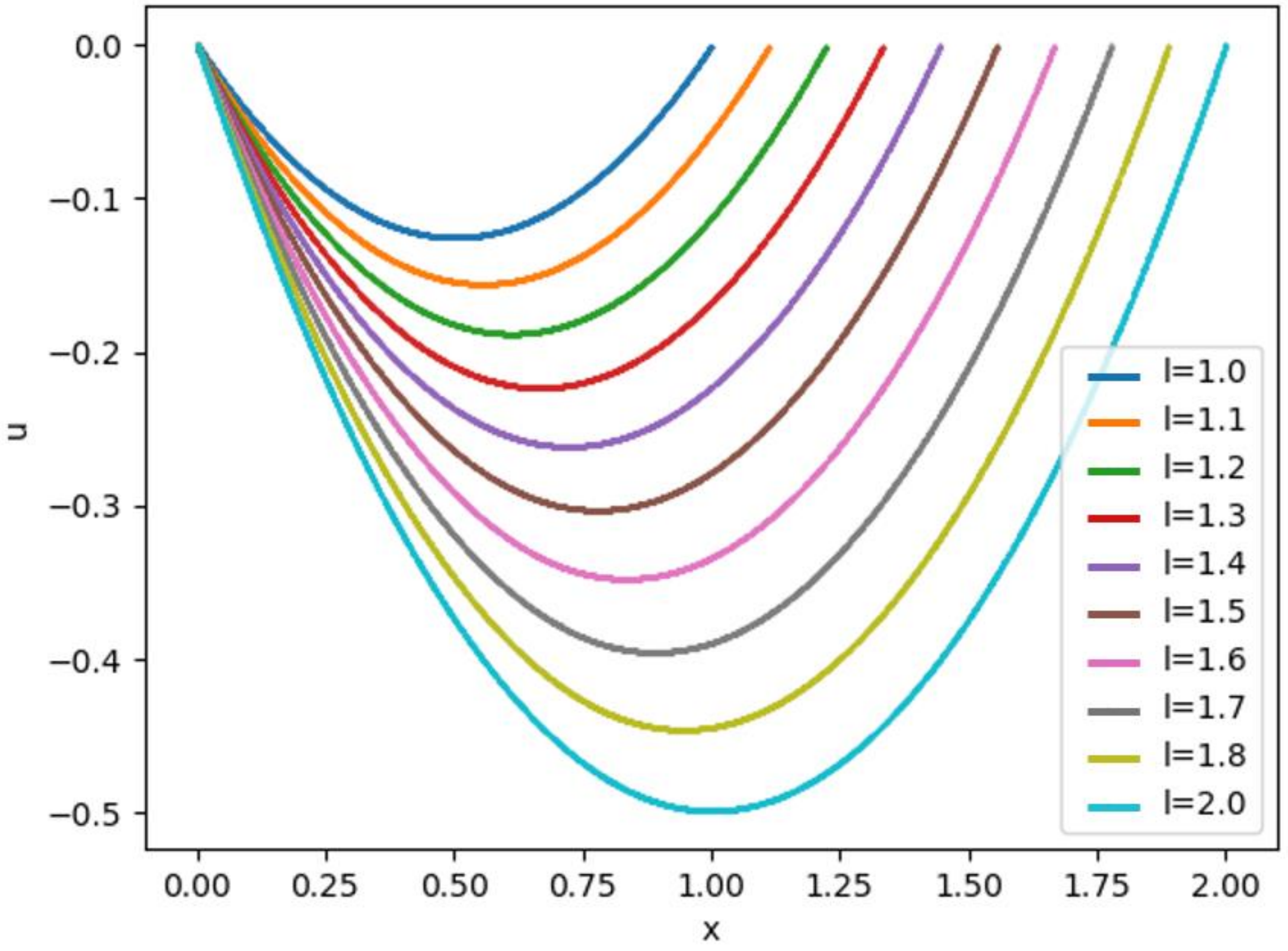
$$L_{Residual} = \int_1^2 \int_0^1 \left(\frac{\delta^2 u_{net}}{\delta x^2}(x) - f(x) \right)^2 dxdl \approx \left(\int_1^2 \int_0^1 dx\, dl \right) \frac{1}{N} \sum_{i=0}^{N} \left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i, l_i) - f(x_i) \right)^2$$

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

Parameterized Problems

Results

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



Solution to the parametric problem

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 $f(x)$

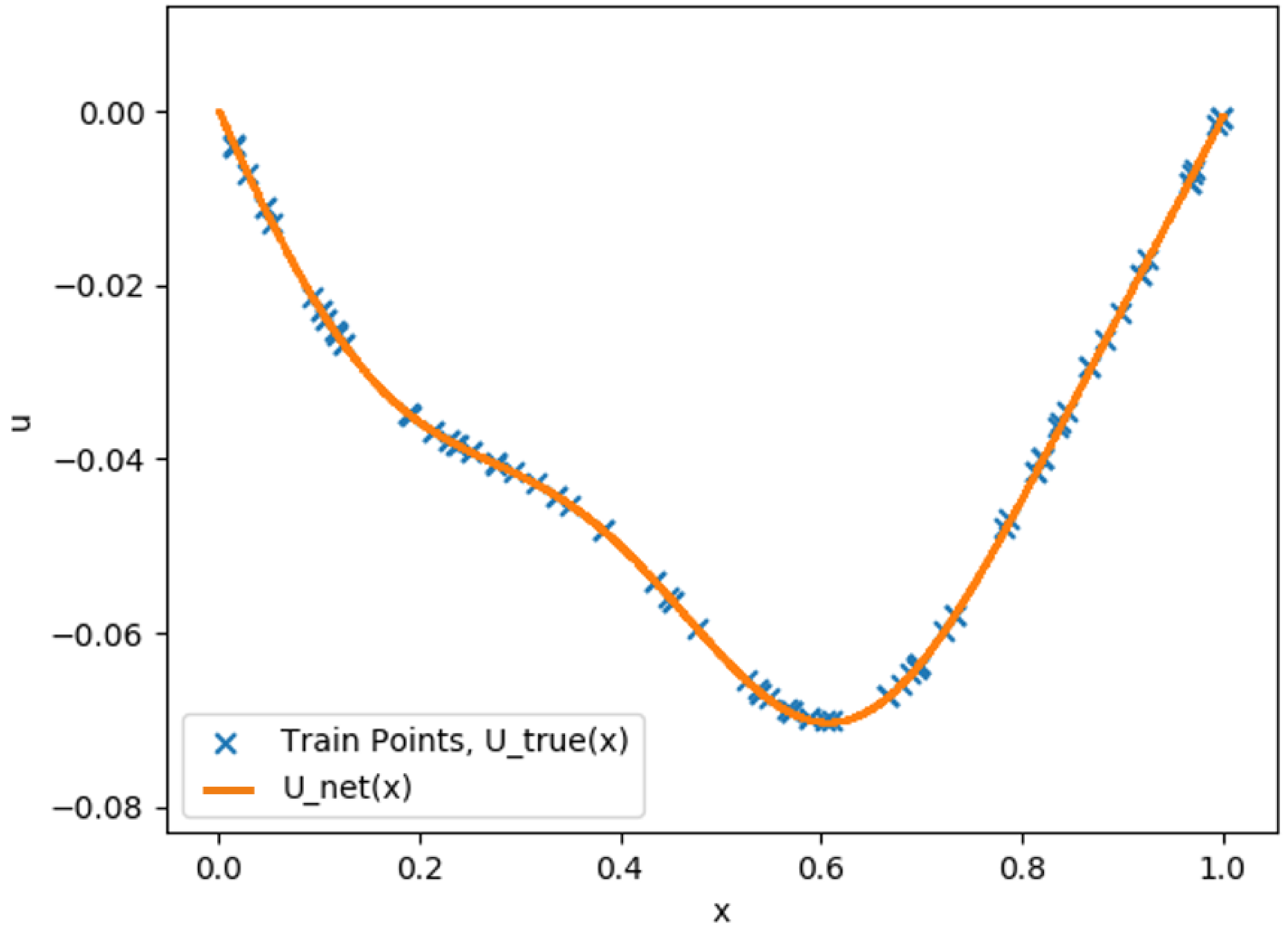
$$L_{Residual} \approx \left(\int_0^1 dx\right) \frac{1}{N} \sum_{i=0}^{N} \left(\frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i)\right)^2$$

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

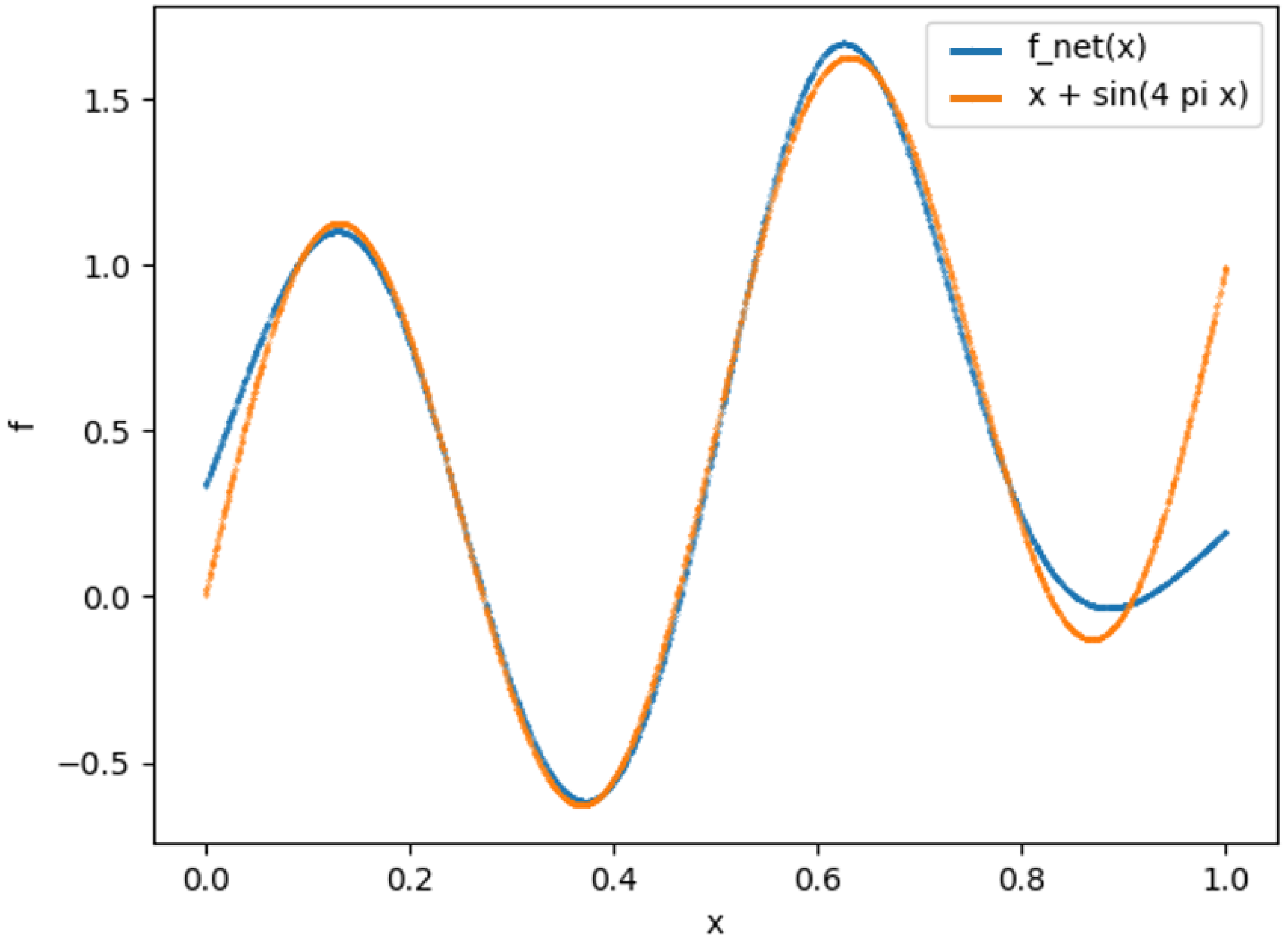
Inverse Problems

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 $x + \sin(4\pi x)$



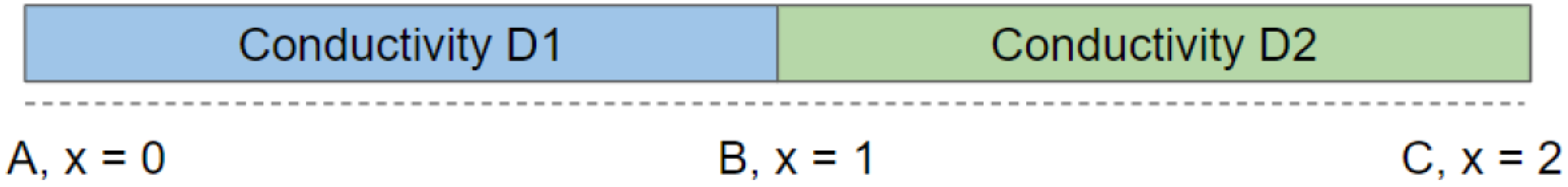
Comparison of $u\_net\ (x)$ and train points from u_true



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

1D diffusion

Problem description



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

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

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

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

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

Solution to 1D diffusion

Code snippets – Custom symbolic PDE

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

```
class Diffusion(PDE):
    name = "Diffusion"
```

Create a child class from physicsnemo' PDE class

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

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

```
        # 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")
```

Symbolic input variables using sympy's `Symbol`

```
        # 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)
```

Dependent variables defined using sympy's `Function`

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

```
        # 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
        )
```

Symbolic PDE. Derivatives are computed using sympy's functions

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

Solution to 1D diffusion

Code snippets

Loading hydra configs

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

Equation and neural network nodes

```
    # 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)]
    )
```

Domain and Constraints

```
    # 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")
```

Sample the boundary of geometry

```
    # 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")
```

Criteria for sub-sampling

```
    # 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")
```

Sample the interior of geometry

```
    # 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")
```

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")
```

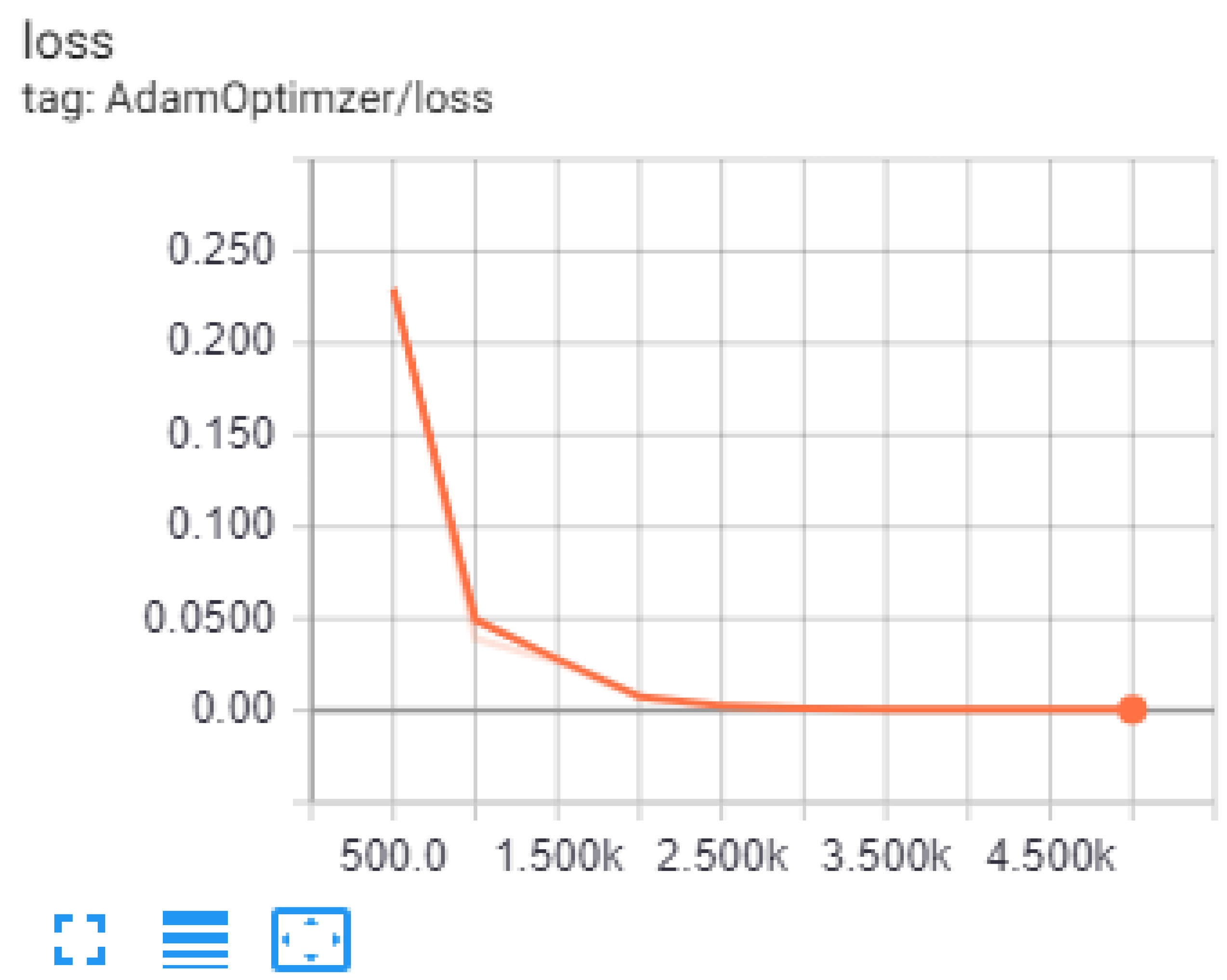
Validators to compare with experimental/analytical /solver data

```
# 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)
```

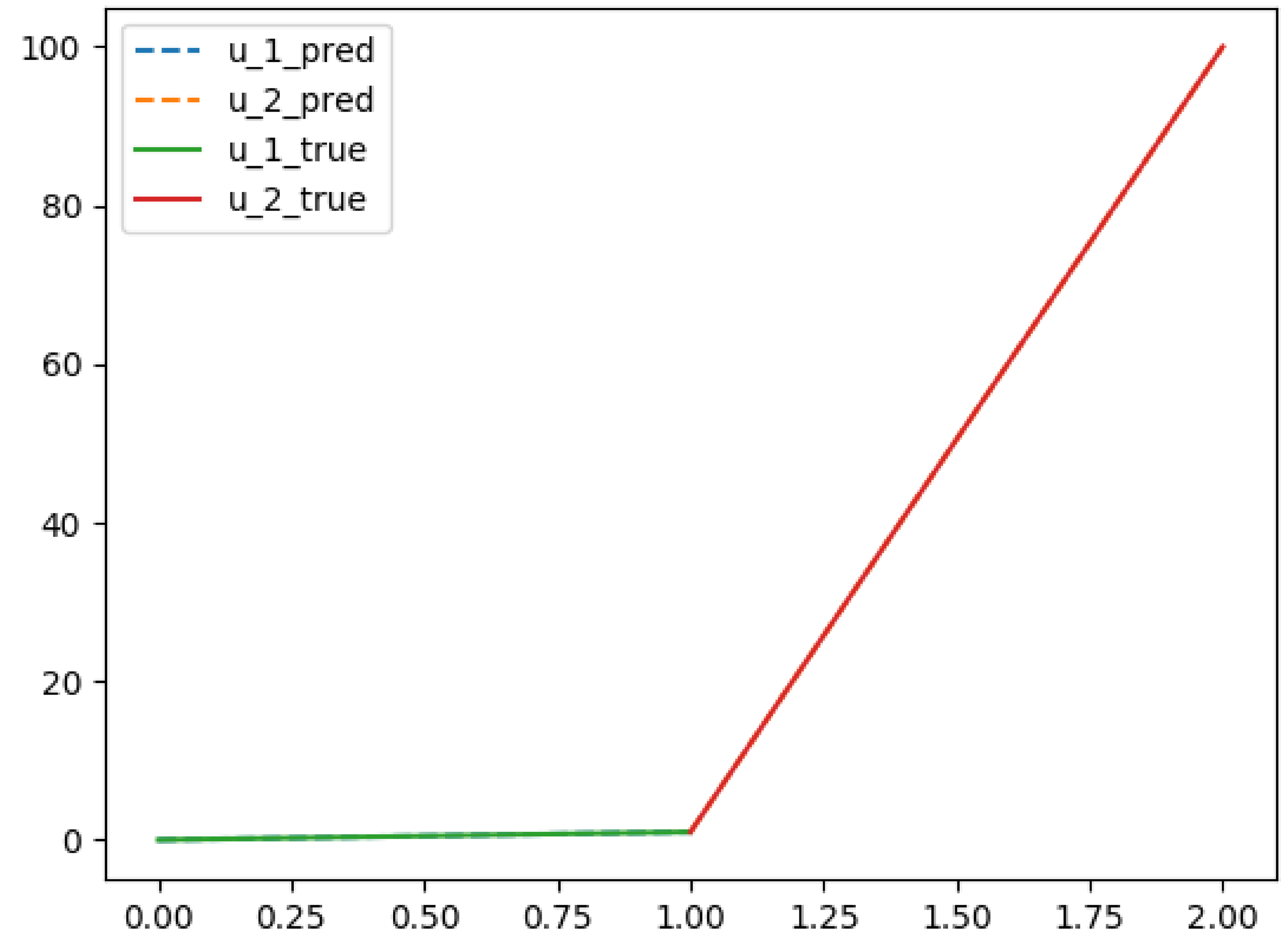
Monitor the quantities of interest during the runtime

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

# start solver
slv.solve()
```

Solver



Tensorboard visualization of loss curves



Results generated from numpy output

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

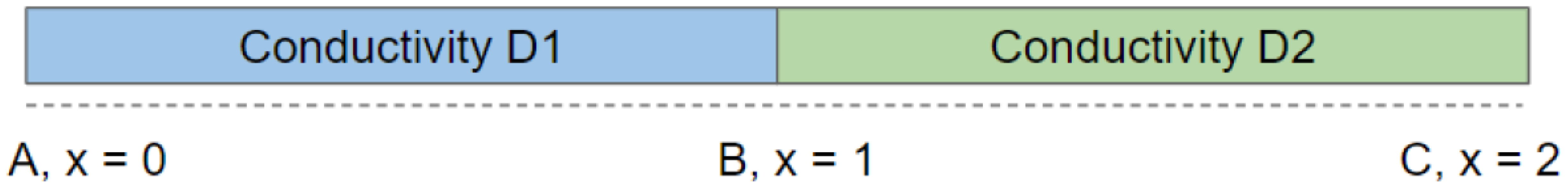
```python
# params for domain
L1 = Line1D(0, 1)
L2 = Line1D(1, 2)

D1 = Symbol("D1")
D1_range = {D1: (5, 25)}
D1_validation = 1e1
```

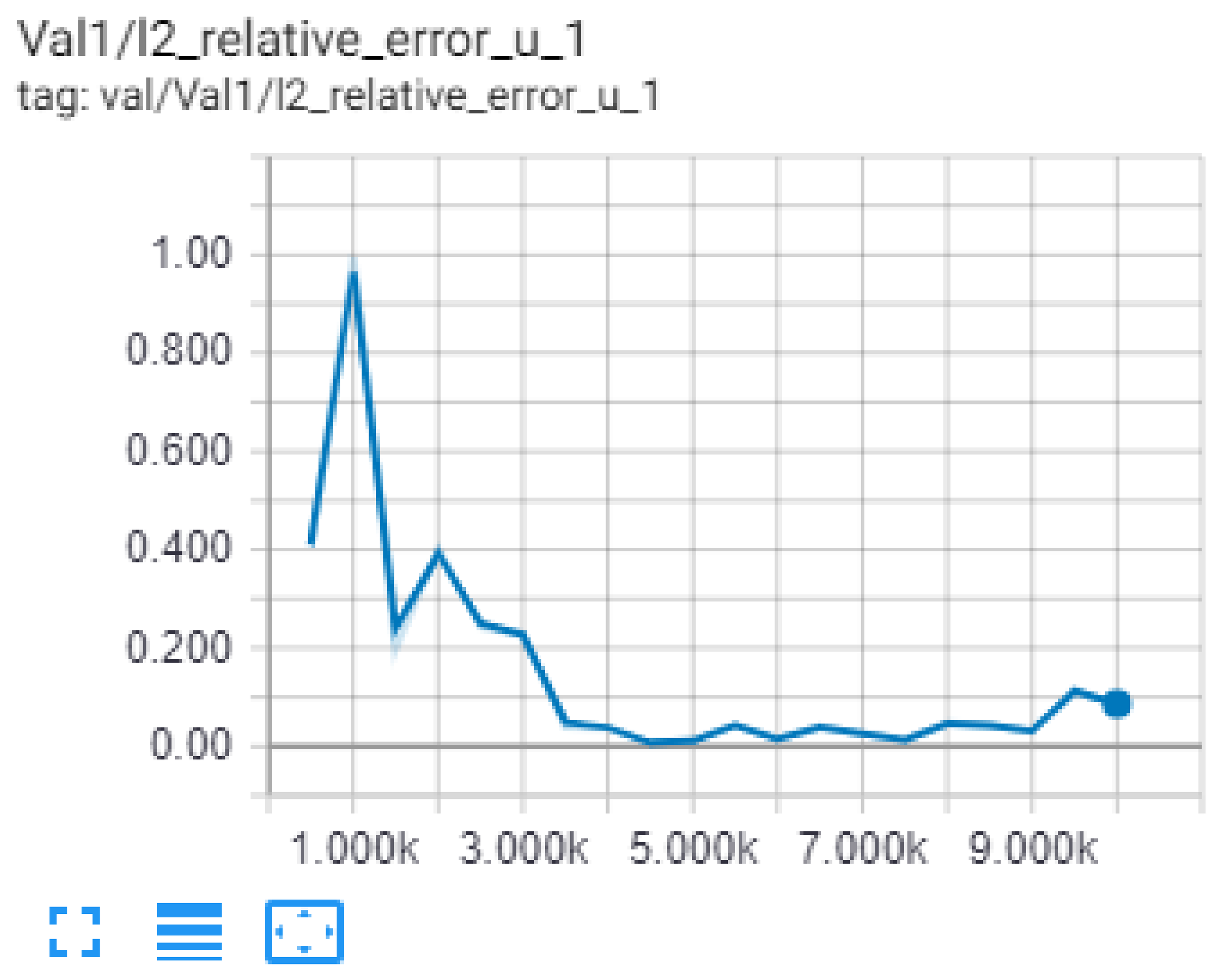
```python
@physicsnemo.sym.main(config_path="conf", config_name="config_param")
def run(cfg: PhysicsNeMoConfig) -> None:

    # 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,
    )
```

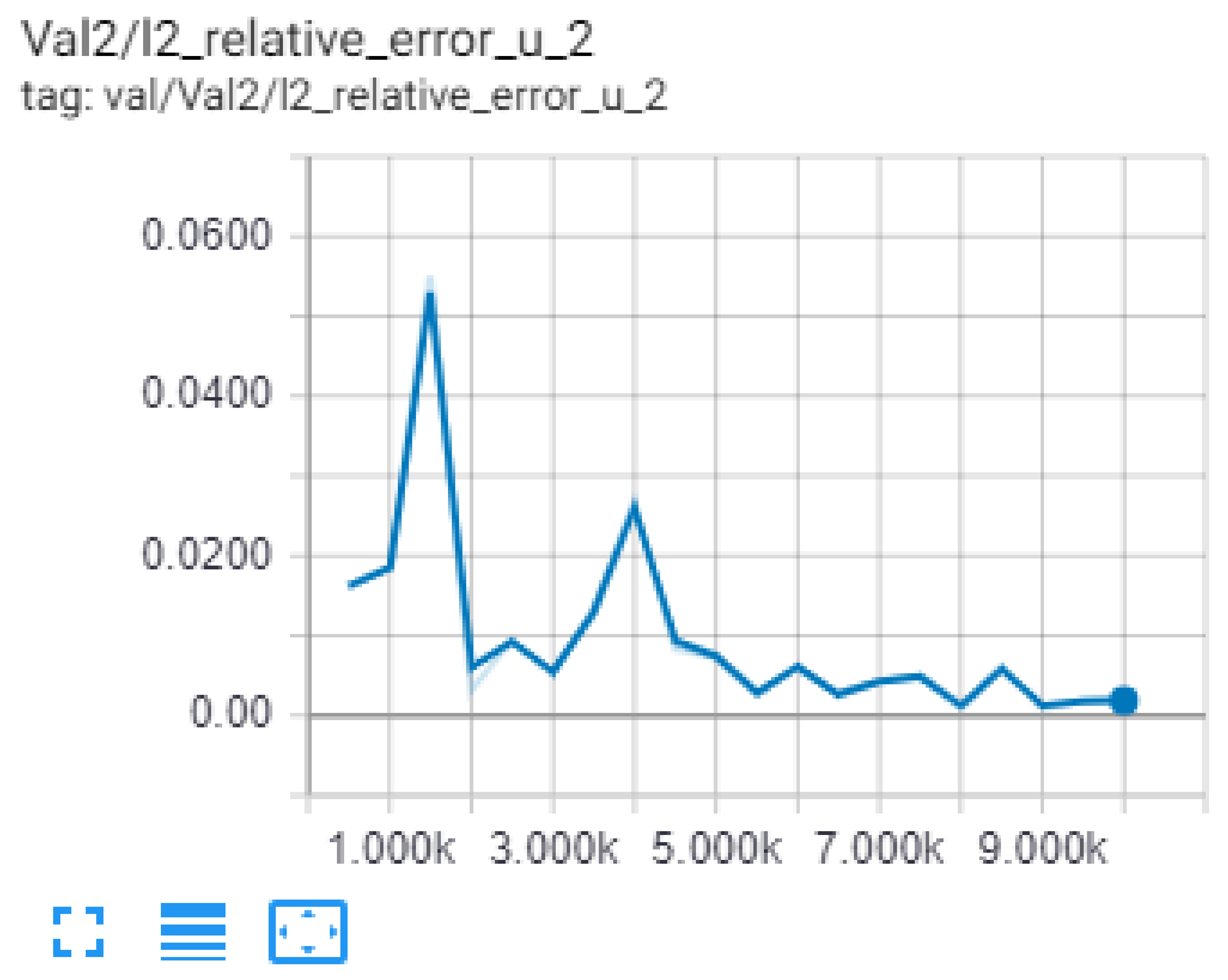
Symbolically parameterize the variables of choice (geometric/physical) and setup the architecture

```python
    # 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")
```

Specify the appropriate parameterization to the constraints





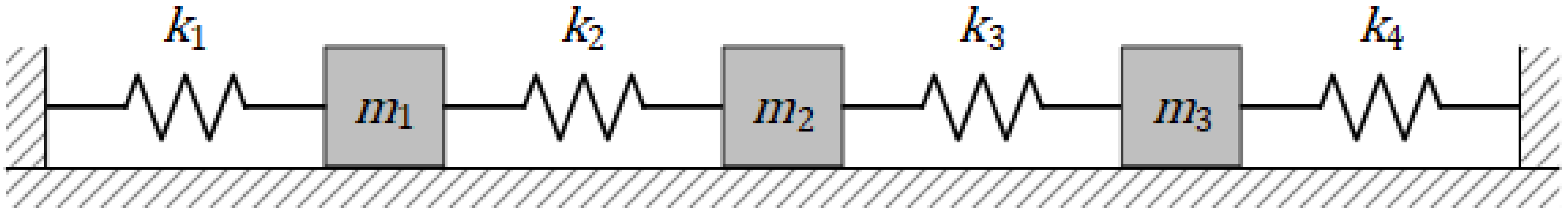


Validation error for $D_1 = 10$

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

Optional - 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

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

Analytical data

```python
    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)})
```

Additional network to invert out the unknowns

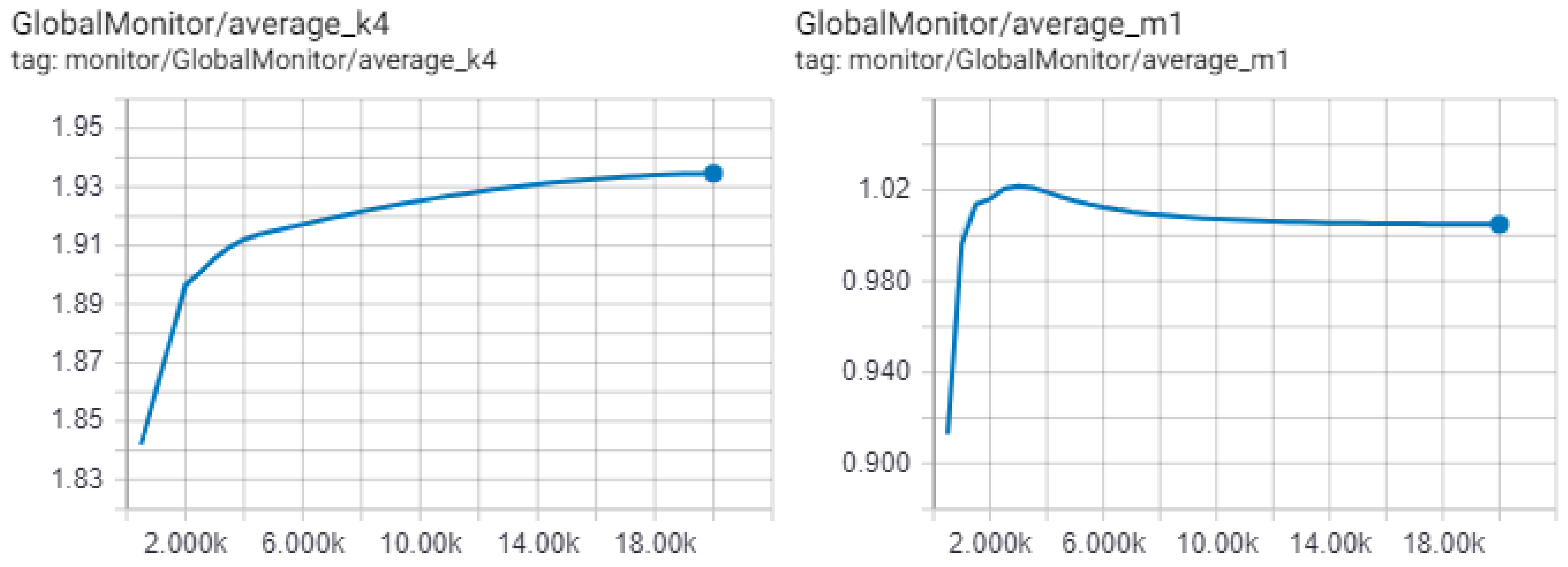
```python
    # make list of nodes to unroll graph on
    sm = SpringMass(k=(2, 1, 1, "k4"), m=("m1", 1, 1))
    sm_net = instantiate_arch(
        input_keys=[Key("t")],
        output_keys=[Key("x1"), Key("x2"), Key("x3")],
        cfg=cfg.arch.fully_connected,
    )
    invert_net = instantiate_arch(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

Assimilate the data using `PointwiseConstraint`

```python
    # 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")
```

Monitors to infer the inverted quantities

```python
    # 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)
```



Results