PHYSICS-INFORMED NEURAL NETWORKS WITH NVIDIA MODULUS: APPLICATION TO EXTERNAL FLOW PROBLEMS

Niki Loppi, Ph.D.

AI & HPC Solutions Architect, HER/NVAITC, NVIDIA

@Teratec Forum, 15 June, 2022

Introduction: What is Modulus?

Credit to: Mohammad Nabian, Ph.D
Senior Software Engineer, AI-HPC, NVIDIA
+ Modulus Team

NVIDIA Modulus

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

Scalable Performance

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

AI Toolkit

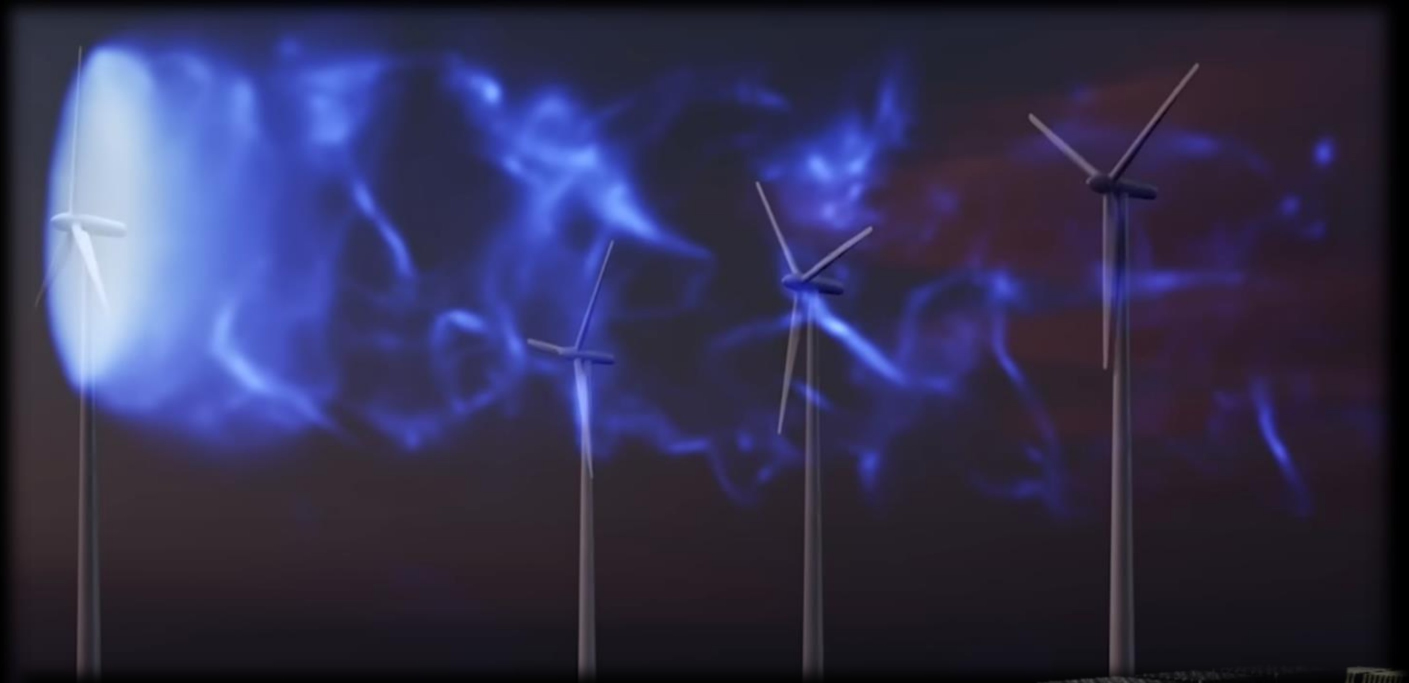
Offers building blocks for developing physics ML surrogate models

Near-Real-Time Inference

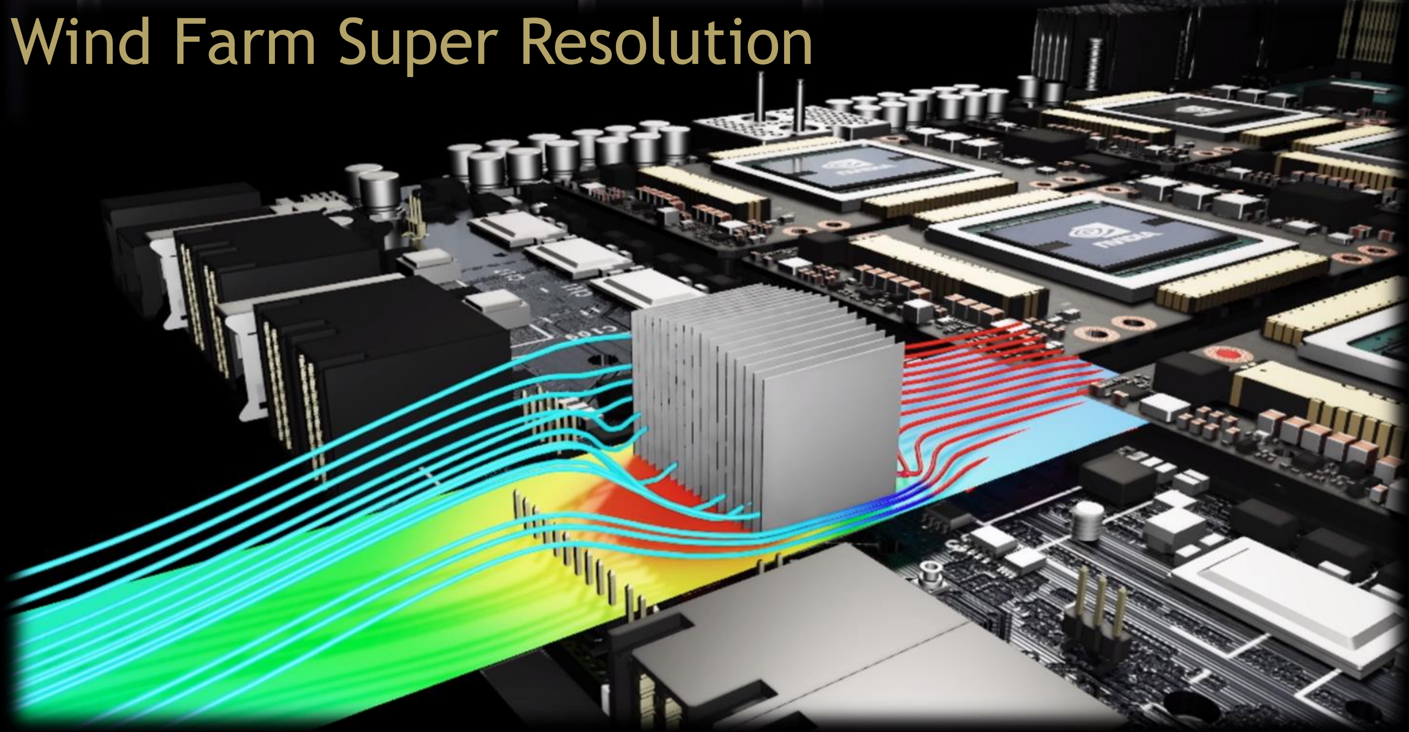
Provides parameterized system representation that solves for multiple scenarios in near real time, trains once offline to infer in real time repeatedly.

Easy to Adopt

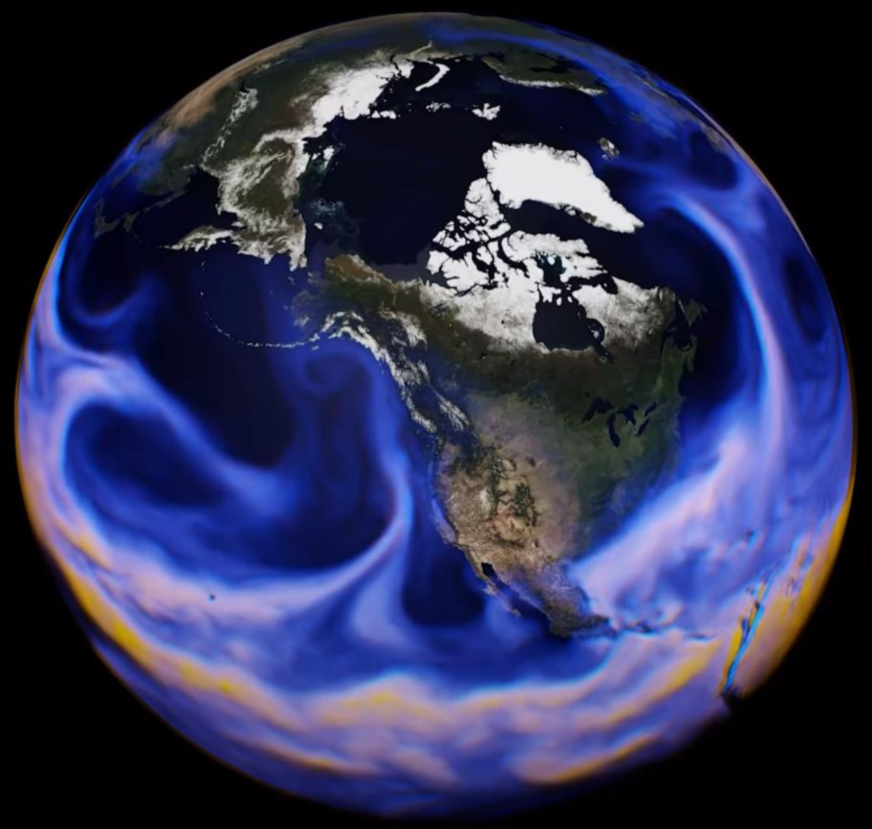
Includes APIs for domain experts to work at a higher level of abstraction. Extensible to new applications with reference applications serving as starting points.



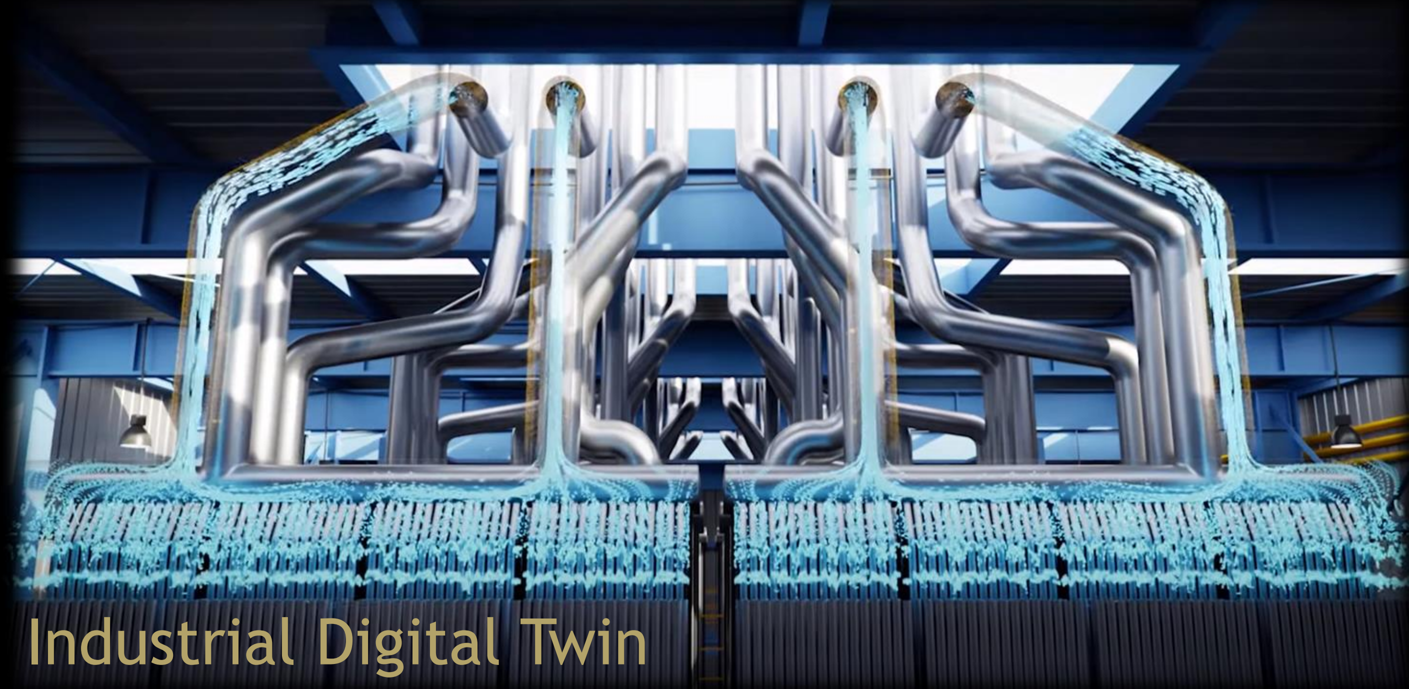
Wind Farm Super Resolution



FPGA Design Optimization



Extreme Weather Prediction

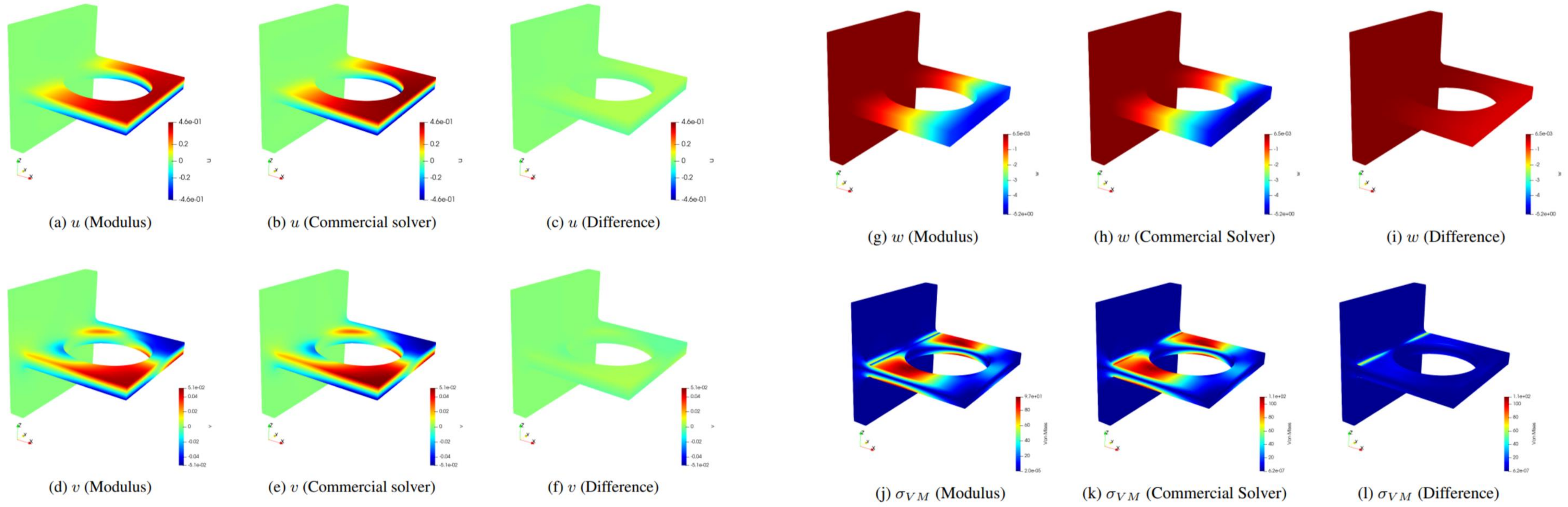


Industrial Digital Twin

What is Modulus?

Modulus is a PDE solver

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



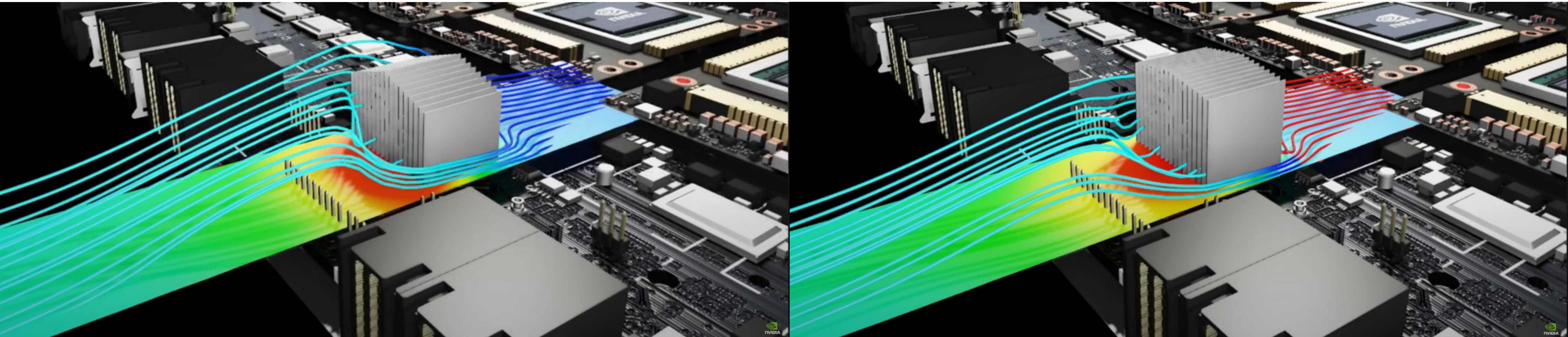
A comparison between Modulus and commercial solver results for a bracket deflection example. Linear elasticity equations are solved here.

What is Modulus?

Modulus is a tool for efficient design optimization & design space exploration

- 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.

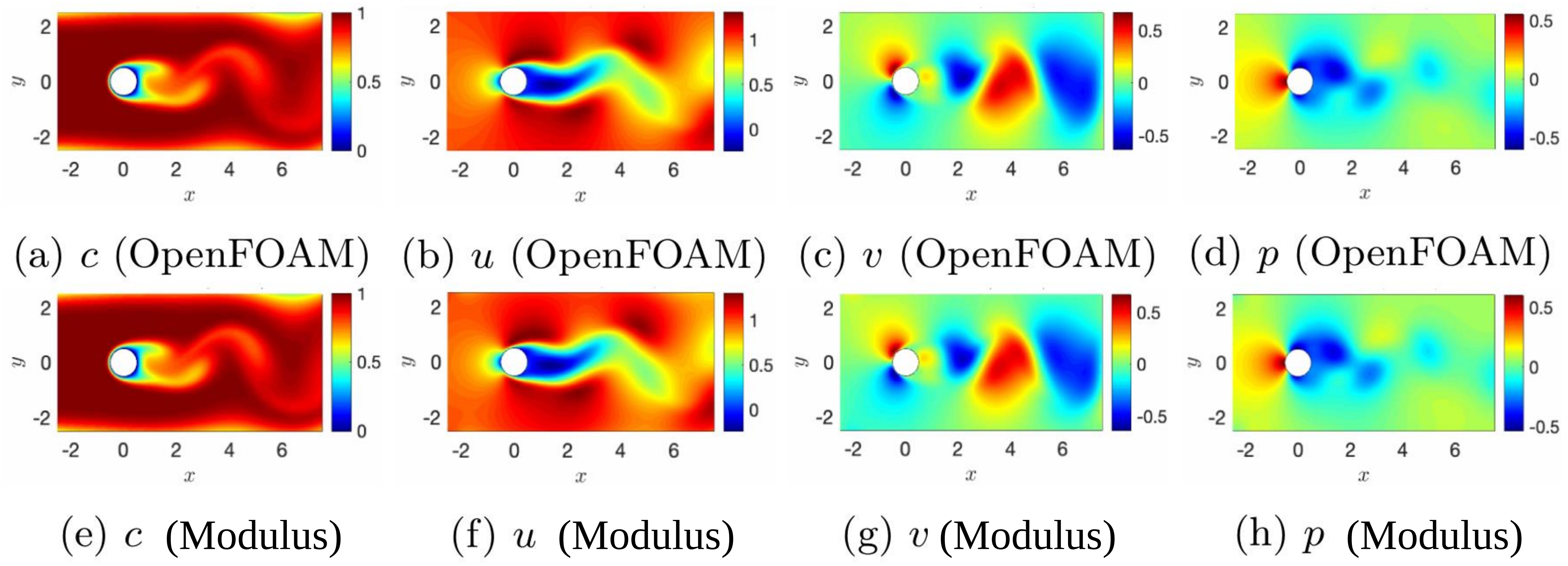
Efficient design space exploration of the heat sink of a Field-Programmable Gate Array (FPGA) using Modulus.



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.

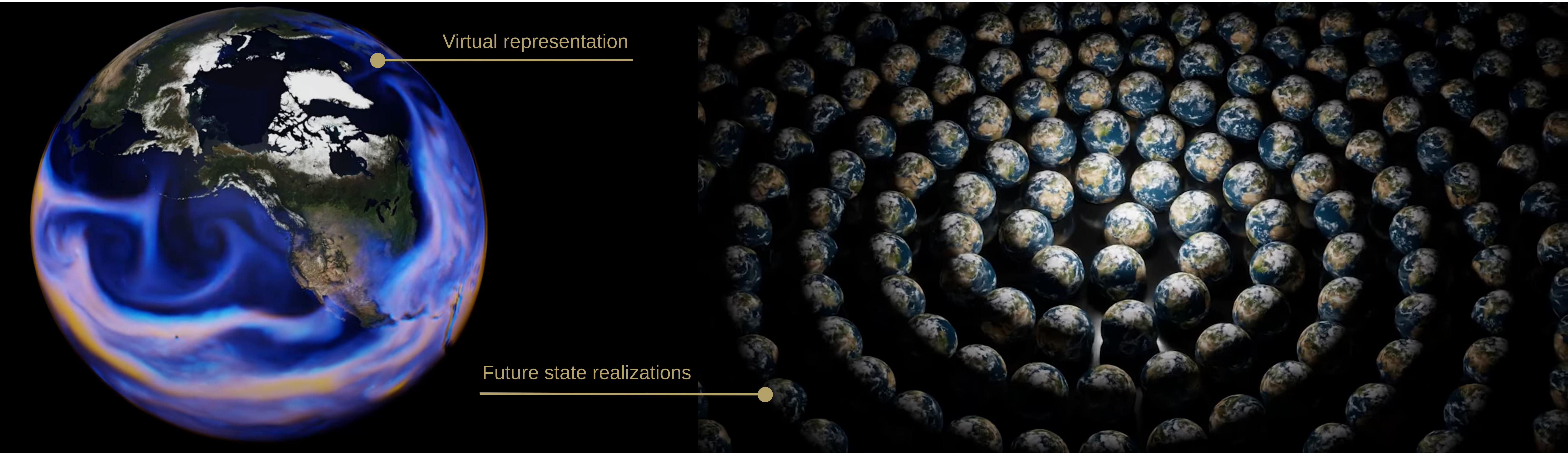


A comparison between Modulus and OpenFOAM results for the flow velocity, pressure and passive scalar concentration fields. Modulus has inferred the velocity and pressure fields using scattered data from passive scalar concentration.

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 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

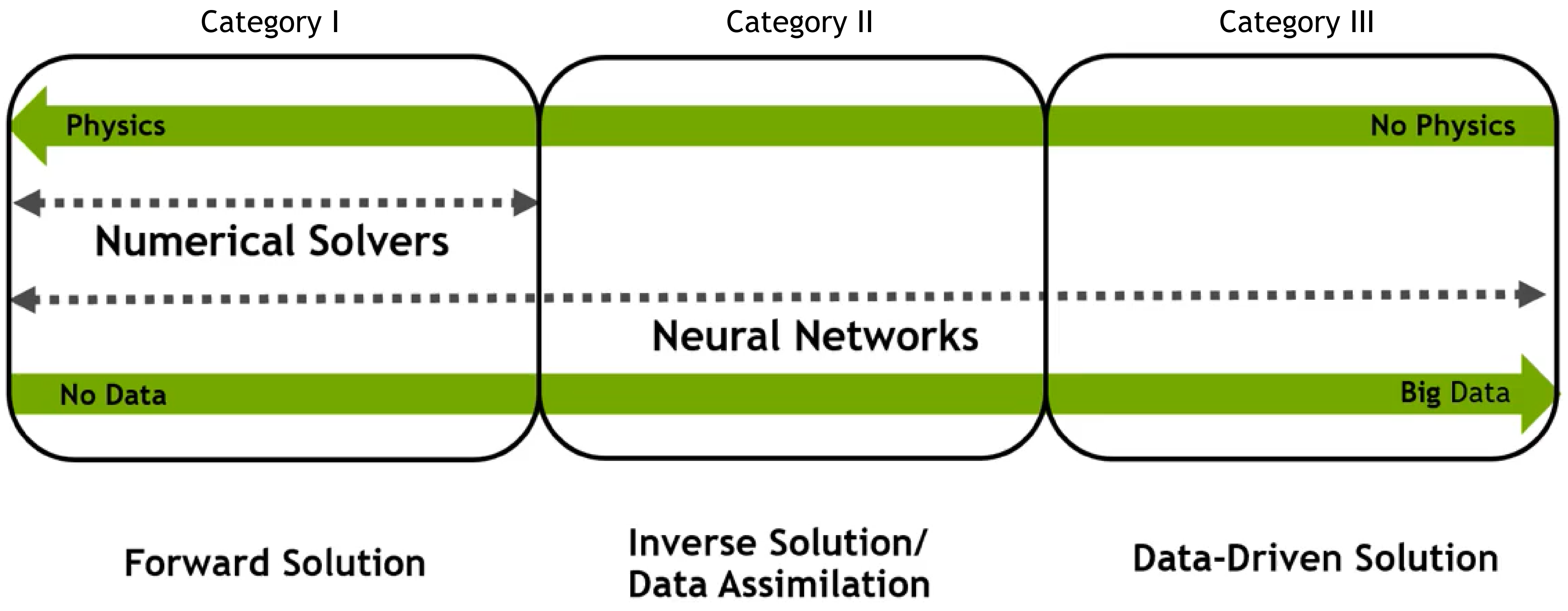
Super-resolution of flow in a wind farm using Modulus.

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 data-driven solutions to engineering problems (category III)

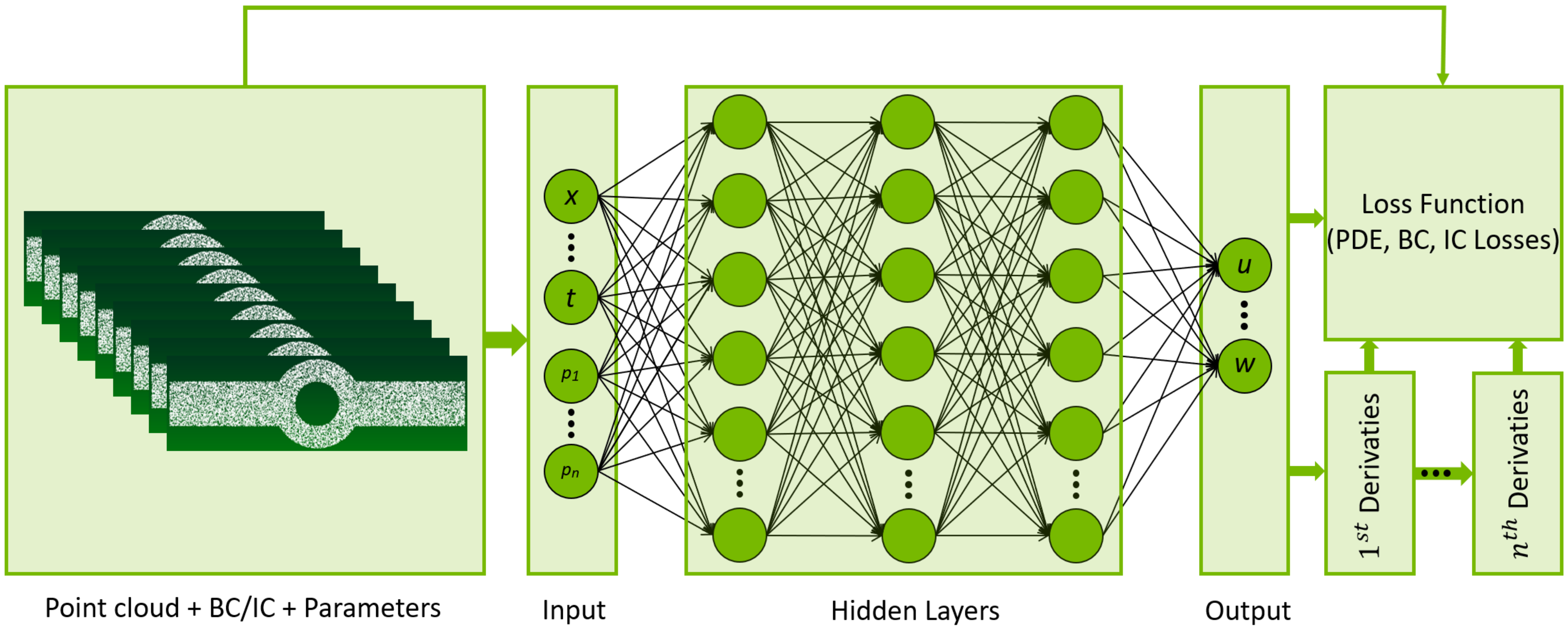
These are all done by developing deep neural network models in Modulus that are physics-informed and/or data-informed.



Physics-Informed Neural Network Solver Methodology

Neural Network Solver Architecture

- A neural network solver approximates the PDE solution using a feed-forward fully-connected neural network.
- The model is trained by constructing a loss function for how well the network satisfies the PDE and constraints.
- If the network can minimize this loss, then it will in effect, solve the given PDE.
- Unlike the data-driven deep learning models, neural network solvers do not require any training data.



MODULUS METHODOLOGY

How Neural Network Solvers Work

The idea is to use a neural network to approximate the solution to given differential equation and boundary conditions.
Example Problem,

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

Construct a deep multi-layer perception $u_{net}(x) \rightarrow u$. $x \in \mathbb{R}$. Assume that $u_{net} \in C^{\infty}$. This means using activation functions like *tanh*, *swish*, *sin*, *sigmoid*... [1]

MODULUS METHODOLOGY

How Neural Network Solvers Work

Construct a Loss function to train $u_{net}(x)$. We can compute $\frac{\delta^2 u_{net}}{\delta x^2}(x)$ using automatic differentiation.

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

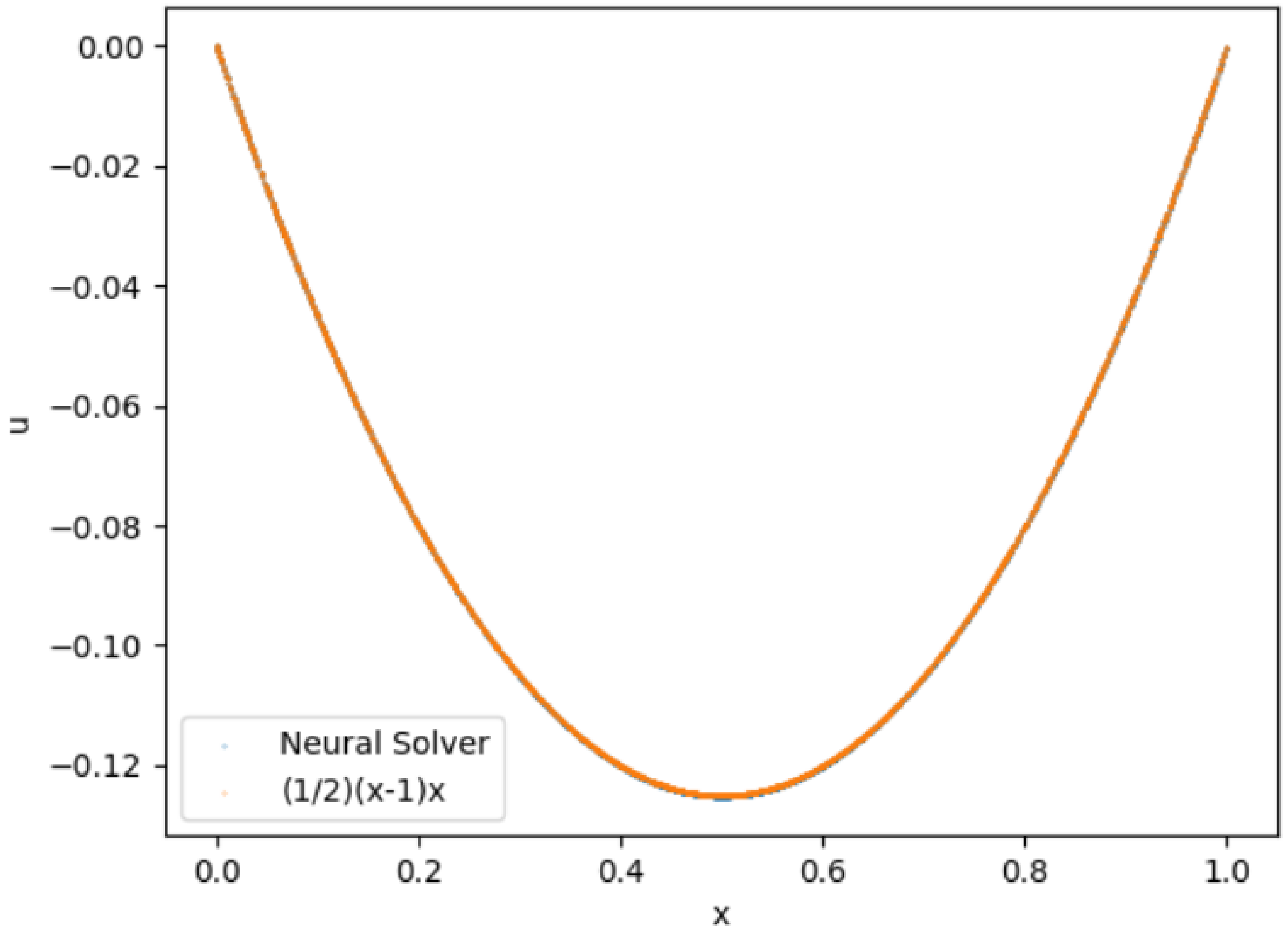
$$L_{residual} = \frac{1}{N}\sum_{i=0}^{N}(\frac{\delta^2 u_{net}}{\delta x^2}(x_i) - f(x_i))^2; x_i \in (0,1) \tag{3}$$

$$L = L_{BC} + L_{residual} \tag{4}$$

MODULUS METHODOLOGY

How Neural Network Solvers Work

$$f(x) = 1$$



NVIDIA Modulus features

PDE Modules

- Modulus includes a collection of PDEs written in symbolic math using Sympy:
  - Diffusion
  - Advection diffusion
  - Navier Stokes
  - Zero-equation & 2-equation turbulence models
  - Linear elasticity
  - Wave equation
  - Electromagnetics
- User can import these PDEs for their examples.
- Alternatively, user can define custom PDEs. Here for example, Poisson and surface flux equations.

```python
from sympy import Symbol, Function
# define Poisson equation and flux with sympy
class SurfacePoisson(PDES):
    name = "SurfacePoisson"

    def __init__(self):
        # represent coordinates & normals in Sympy symbolic form
        x, y, z = Symbol("x"), Symbol("y"), Symbol("z")
        normal_x, normal_y, normal_z = (
            Symbol("normal_x"),
            Symbol("normal_y"),
            Symbol("normal_z"),
        )

        # Represent the solution u as a Sympy function of spatial coordinates
        u = Function("u")(x, y, z)

        # set Poisson & flux equations in Sympy symbolic form
        # u.diff(x, 2) is second derivative of u w.r.t. x.
        self.equations = {}
        self.equations["poisson_u"] = u.diff(x, 2) + u.diff(y, 2) + u.diff(z, 2)
        self.equations["flux_u"] = (
            normal_x * u.diff(x) + normal_y * u.diff(y) + normal_z * u.diff(z)
        )
```

Neural Network Modules

```
from modulus.hydra import instantiate_arch
from modulus.key import Key

poisson_net = instantiate_arch(
    input_keys=[Key("x"), Key("y"), Key("z")],
    output_keys=[Key("u")],
    cfg=cfg.arch.fully_connected,
)
```

- Modulus includes a collection of neural network architectures, including:
    - Fully connected network
    - Variations of Fourier feature networks
    - Sinusoidal Representation network
    - Deep Galerkin network
    - Multiplicative filter networks
    - Hash encoding network
    - DeepONet
    - Variations of Fourier neural operators
    - Super resolution network
    - Pix2Pix network

- User can import these architectures for their examples.
- Alternatively, user can define custom architectures.

Geometry Modules

Constructive Solid Geometry (CSG) Module

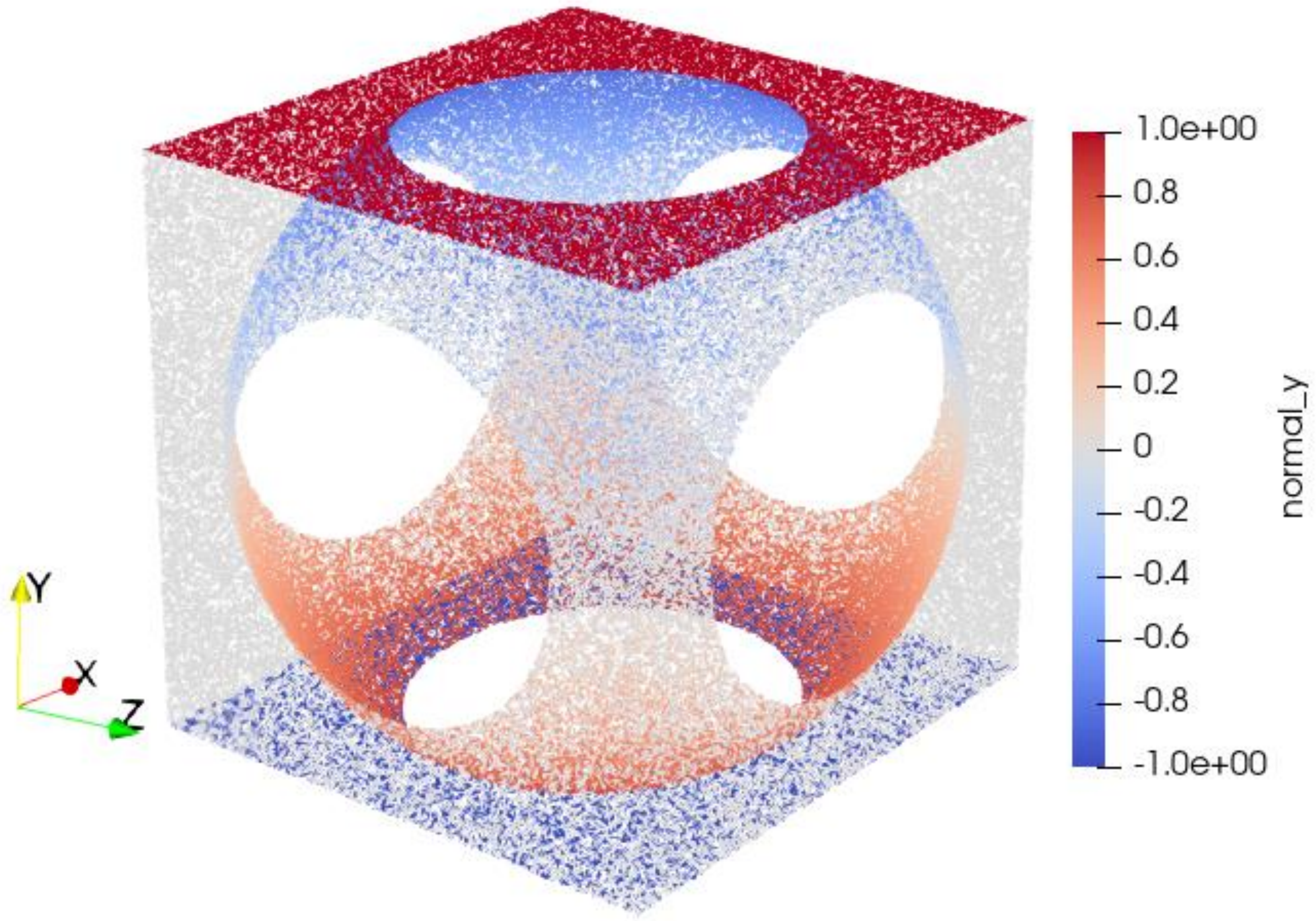
```
from modulus.geometry.csg.csg_3d import Sphere, Box
from modulus.plot_utils.vtk import var_to_polyvtk

# define geometry
sphere = Sphere((0, 0, 0), 1.2)
box = Box((-1, -1, -1), (1, 1, 1))
geo = box - sphere
surface_points = geo.sample_boundary(1024 * 256)
var_to_polyvtk(surface_points, "csg_example")
```

- Allows to create object primitives and perform Boolean operations.
- Also computes SDF, its derivatives, and surface normals.
- Once the geometry is defined, can create a point cloud for training.

- Supported geometry primitives:
  - 1D: Line
  - 2D: Line, rectangle, circle, triangle, ellipse
  - 3D: Plane, box, sphere, cylinder, torus, cone, etc.

- Supported Boolean operations:
  - Union
  - Intersection
  - Subtraction

- Other functionalities:
  - Transform (translation, rotation, scaling)
  - Repeat
  - Sample point cloud on boundary or interior

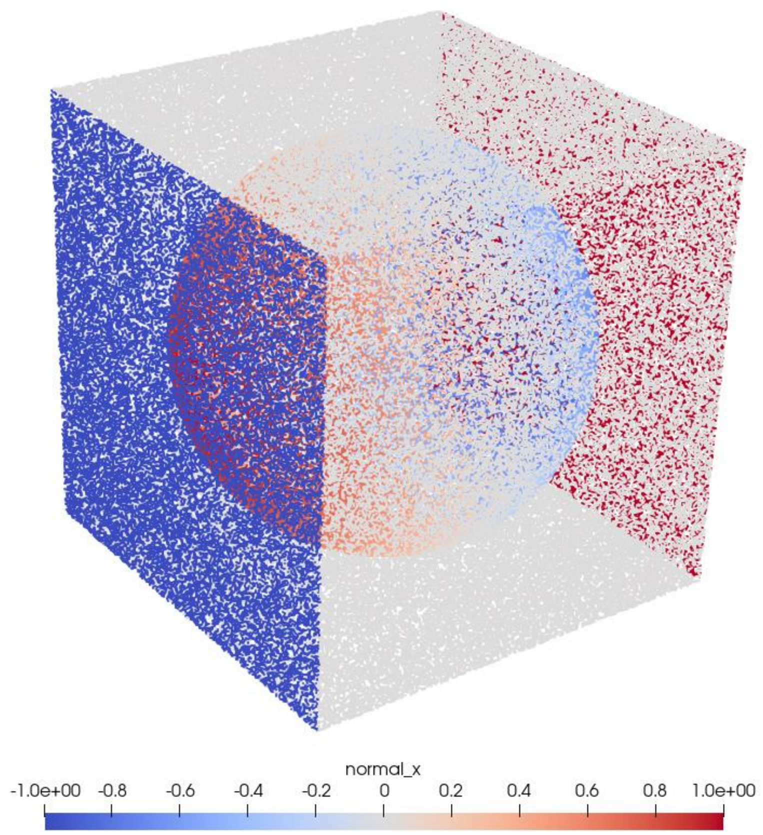


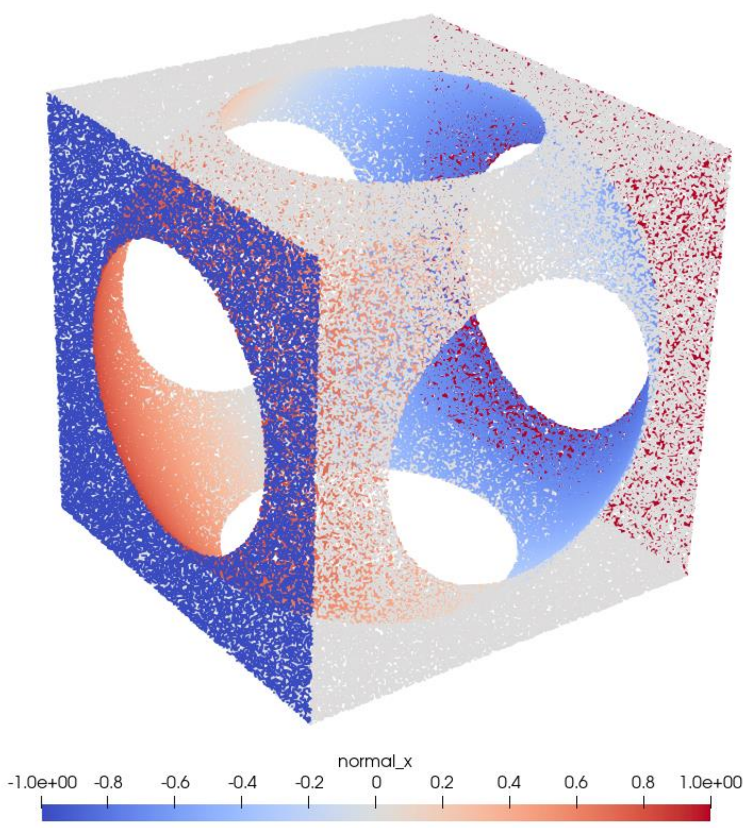
Geometry Modules

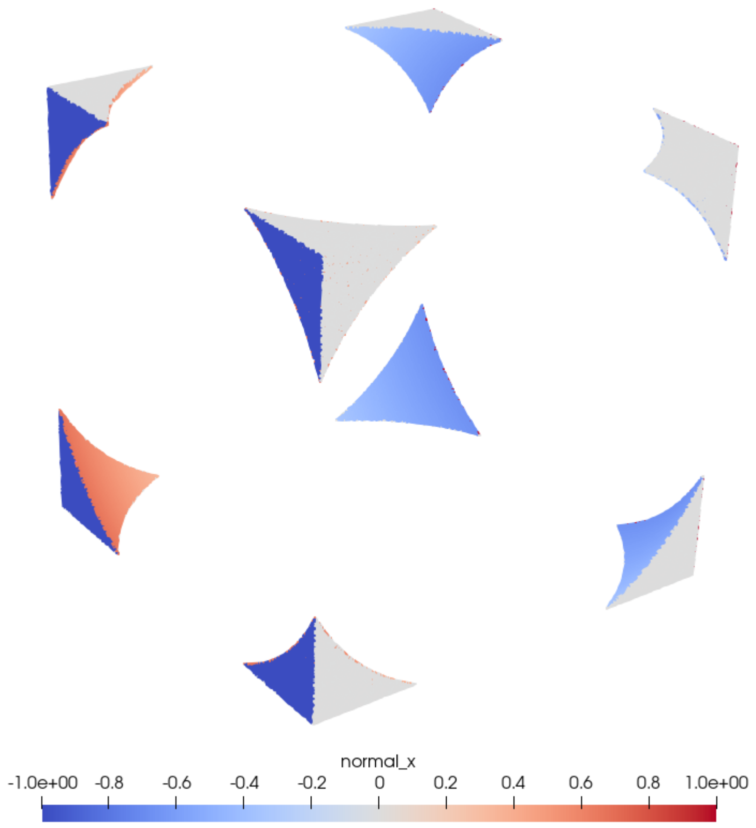
CSG Module- Geometry Parameterization

- CSG module allows creation of parameterized geometries with SymPy

```
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: (0.8, 1.5)}
sphere = Sphere((0, 0, 0), radius)
box = Box((-1, -1, -1), (1, 1, 1))
geo = box - sphere

for i in range(3):
    specific_radius = 0.9 + i * 0.3
    surface_points = geo.sample_boundary(
        1024 * 256, param_ranges={radius: specific_radius}
    )
    var_to_polyvtk(surface_points, "csg_parameterized_example_" + str(i))
```





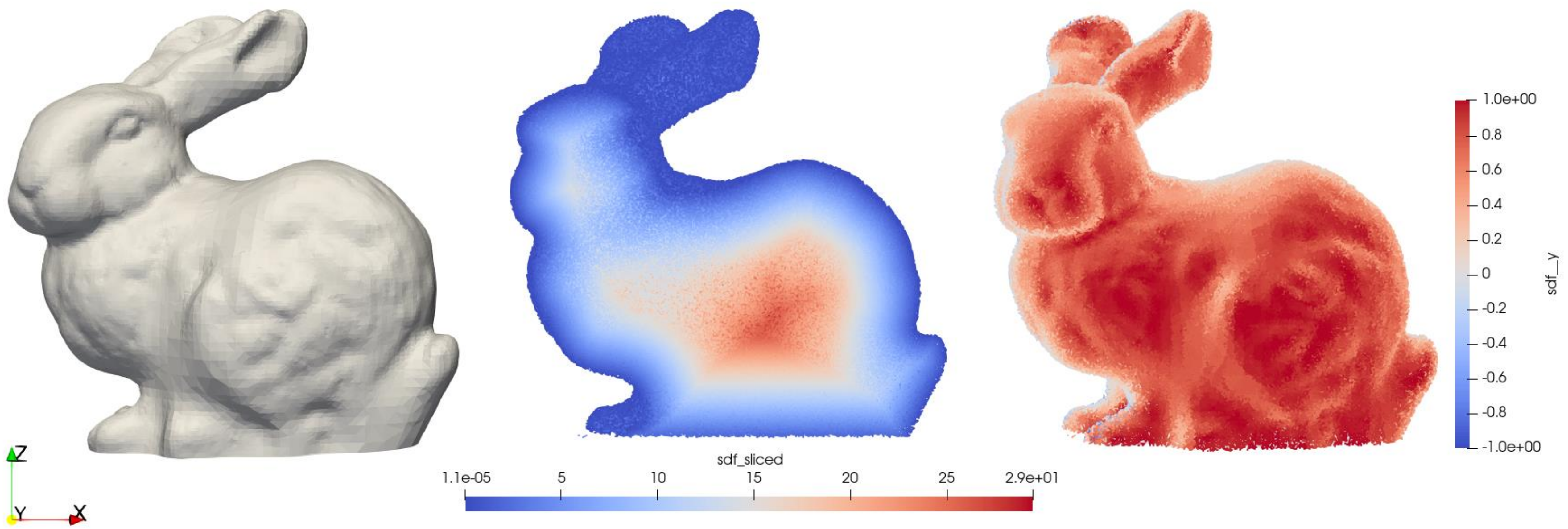


Geometry Modules

Tessellated Geometry (TG) Module

```python
# stl source: https://commons.wikimedia.org/wiki/File:Stanford_Bunny.stl
from modulus.geometry.tessellation.tessellation import Tessellation
from modulus.plot_utils.vtk import var_to_polyvtk

# read stl files to make geometry
geo = Tessellation.from_stl("Stanford_Bunny.stl", airtight=True)

interior_points = geo.sample_interior(1024 * 1024, compute_distance_field=True)
var_to_polyvtk(interior_points, "tg_example")
```

- 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.



Constraint Modules

- In Modulus, different loss terms are defined via constraints.
- Modulus contains various types of constraints:
  - `PointwiseBoundaryConstraint`
  - `PointwiseInteriorConstraint`
  - `IntegralConstraint`
  - `IntegralBoundaryConstraint`
  - `VariationalConstraint`

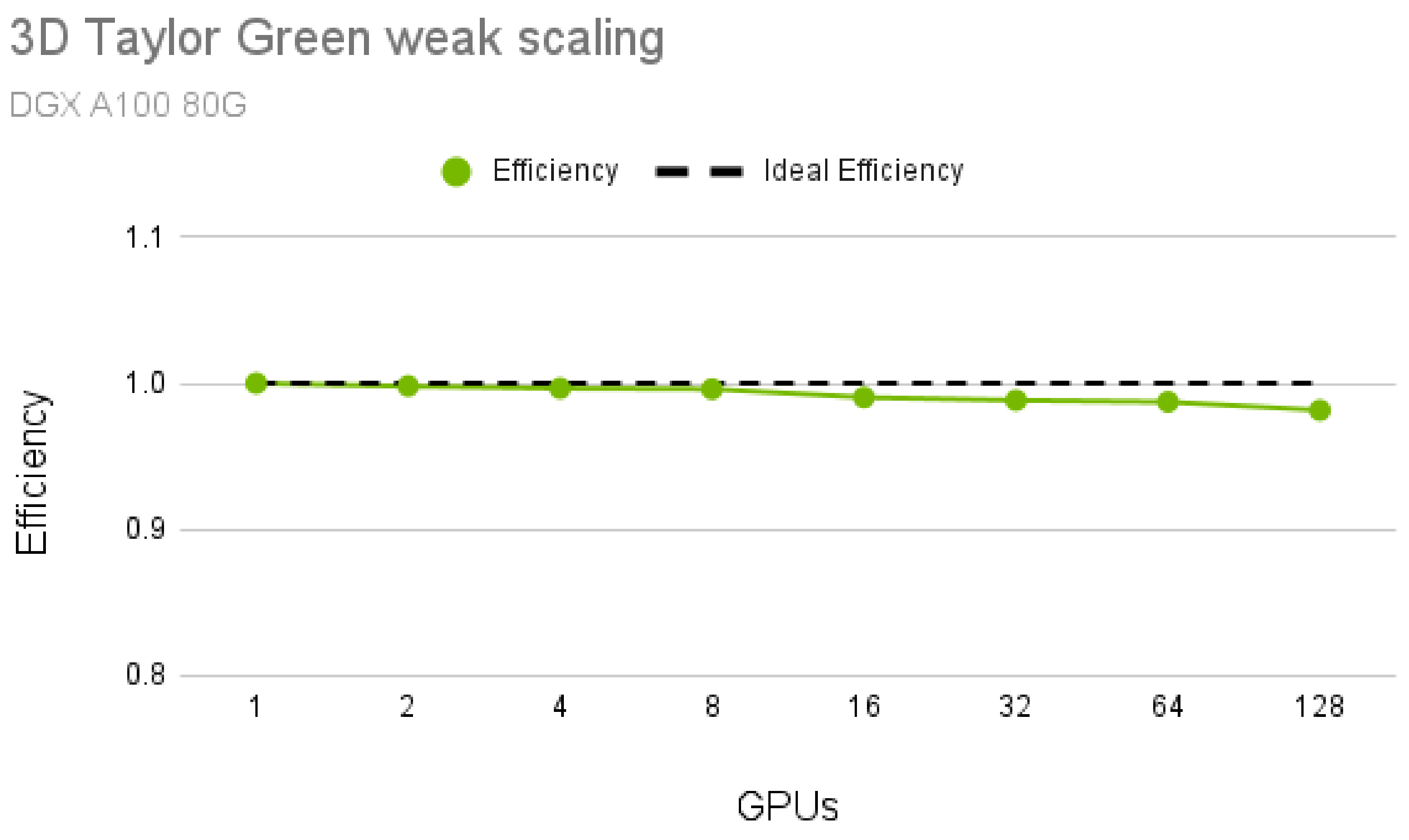
```python
from modulus.continuous.domain.domain import Domain
from modulus.continuous.constraints.constraint import (
    PointwiseBoundaryConstraint,
)

# make domain
domain = Domain()

# sphere surface
surface = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=geo,
    outvar={"poisson_u": -18.0 * x * y * z, "flux_u": 0},
    batch_size=cfg.batch_size.surface,
)
domain.add_constraint(surface, "surface")
```

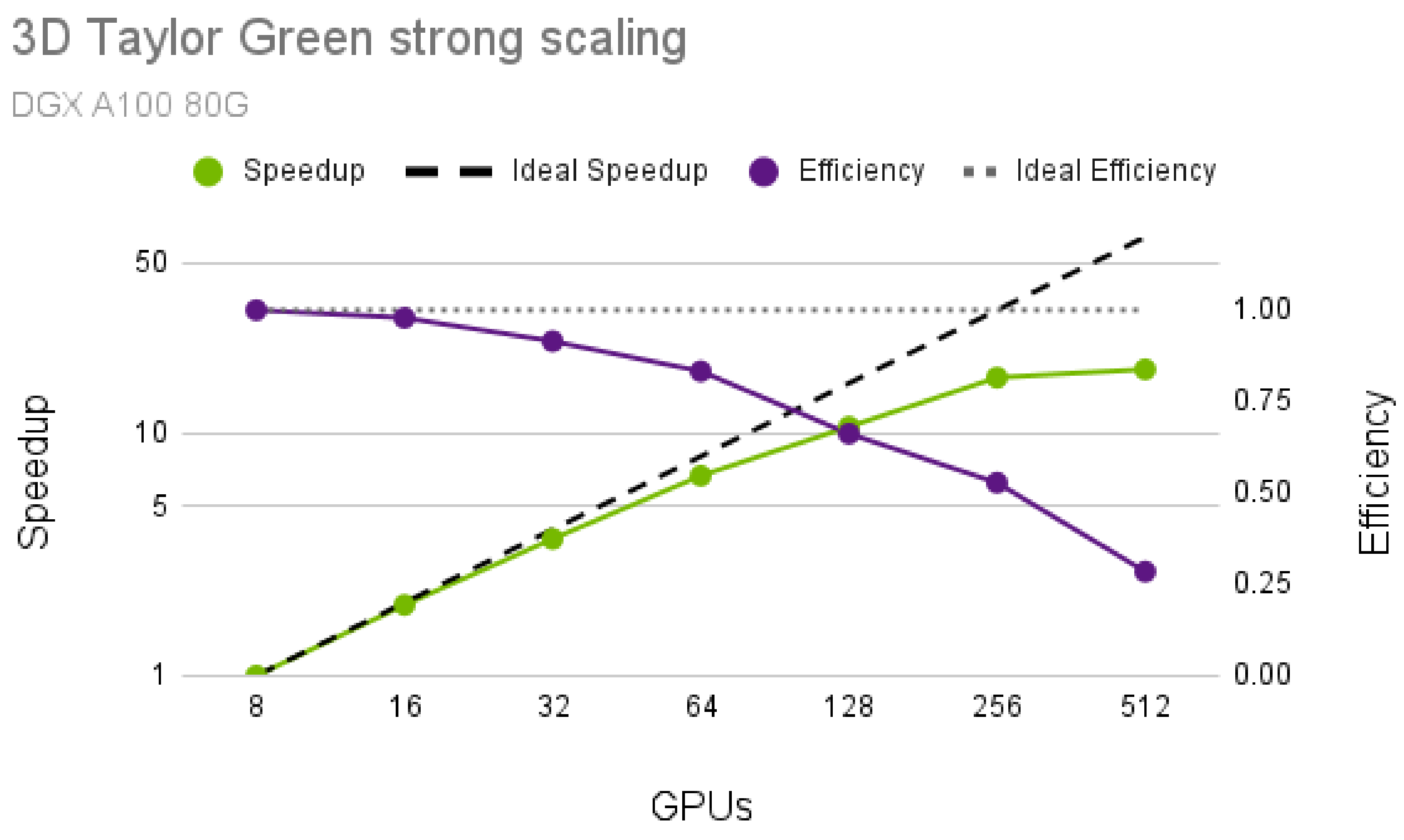
```python
from sympy import Symbol, Function

import modulus
from modulus.hydra import to_yaml, instantiate_arch, to_absolute_path
from modulus.hydra.config import ModulusConfig
from modulus.continuous.solvers.solver import Solver
from modulus.continuous.domain.domain import Domain
from modulus.continuous.constraints.constraint import (
    PointwiseBoundaryConstraint,
)
from modulus.geometry.tessellation.tessellation import Tessellation
from modulus.key import Key
from modulus.pdes import PDES

# define Poisson equation and flux with sympy
class SurfacePoisson(PDES):
    name = "SurfacePoisson"

def __init__(self):
    # represent coordinates & normals in Sympy symbolic form
    x, y, z = Symbol("x"), Symbol("y"), Symbol("z")
    normal_x, normal_y, normal_z = (
    Symbol("normal_x"),
    Symbol("normal_y"),
    Symbol("normal_z"),
    )

    # Represent the solution u as a Sympy function of spatial coordinates
    u = Function("u")(x, y, z)

    # set Poisson & flux equations in Sympy symbolic form
    # u.diff(x, 2) is second derivative of u w.r.t. x.
    self.equations = {}
    self.equations["poisson_u"] = u.diff(x, 2) + u.diff(y, 2) + u.diff(z, 2)
    self.equations["flux_u"] = (
    normal_x * u.diff(x) + normal_y * u.diff(y) + normal_z * u.diff(z)
    )
```

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

    # make list of nodes to unroll graph on
    sp = SurfacePoisson()
    poisson_net = instantiate_arch(
    input_keys=[Key("x"), Key("y"), Key("z")],
    output_keys=[Key("u")],
    cfg=cfg.arch.fully_connected,
    )
    nodes = sp.make_nodes() + [
    poisson_net.make_node(name="poisson_network", jit=cfg.jit)
    ]

    # add constraints to solver
    # make geometry
    x, y, z = Symbol("x"), Symbol("y"), Symbol("z")
    geo = Tessellation.from_stl(to_absolute_path("Stanford_Bunny.stl"), airtight=True)
    geo.scale(0.01)

    # make domain
    domain = Domain()

    # sphere surface
    surface = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=geo,
        outvar={"poisson_u": -18.0 * x * y * z, "flux_u": 0},
        batch_size=cfg.batch_size.surface,
    )
    domain.add_constraint(surface, "surface")

    # make solver
    slv = Solver(cfg, domain)
    # start solver
    slv.solve()

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

PERFORMANCE

MULTI-GPU/NODE Scalability (TensorFlow version)





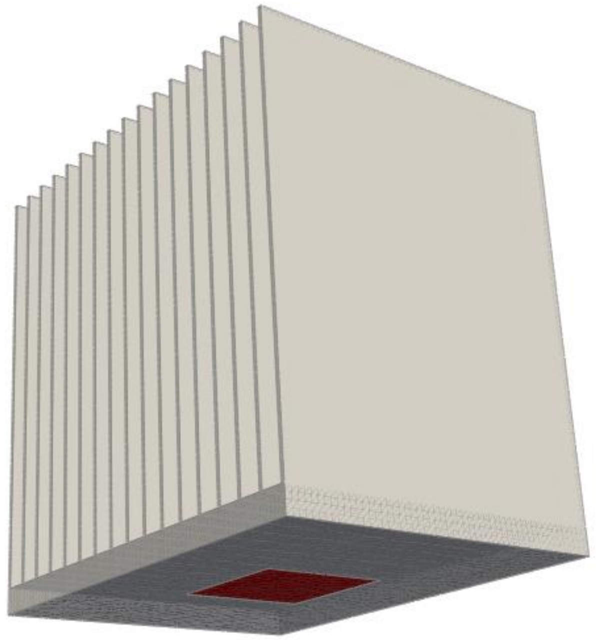
External flow applications

Conjugate heat transfer

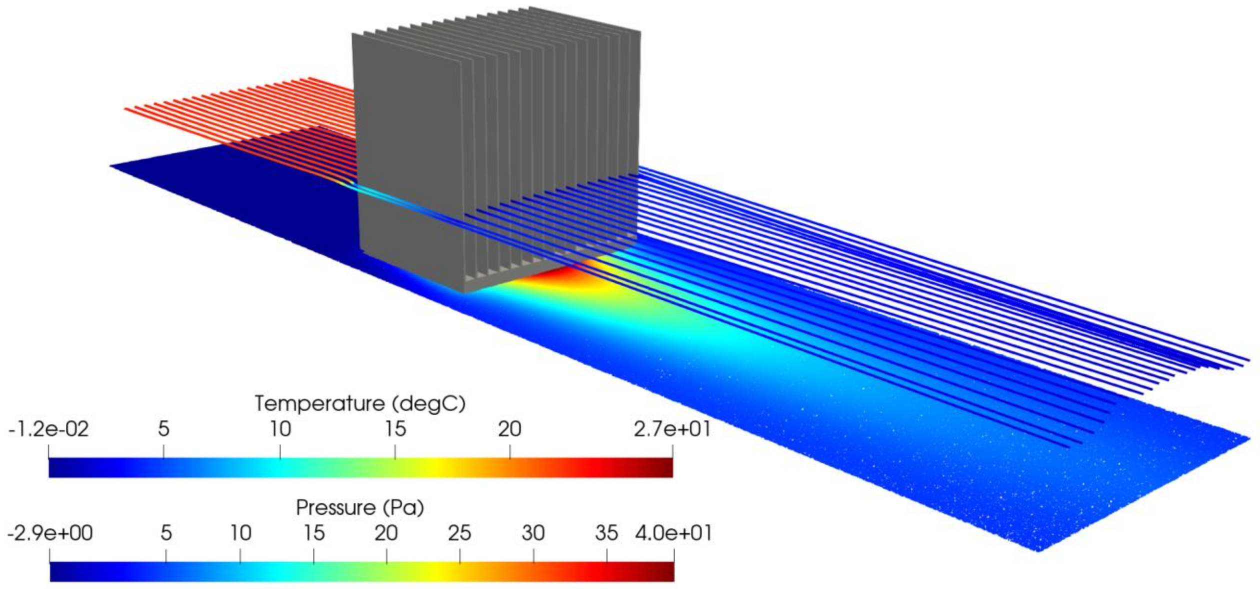
Demonstrating the ability of Modulus to solve multi-physics problems involving high Re flows

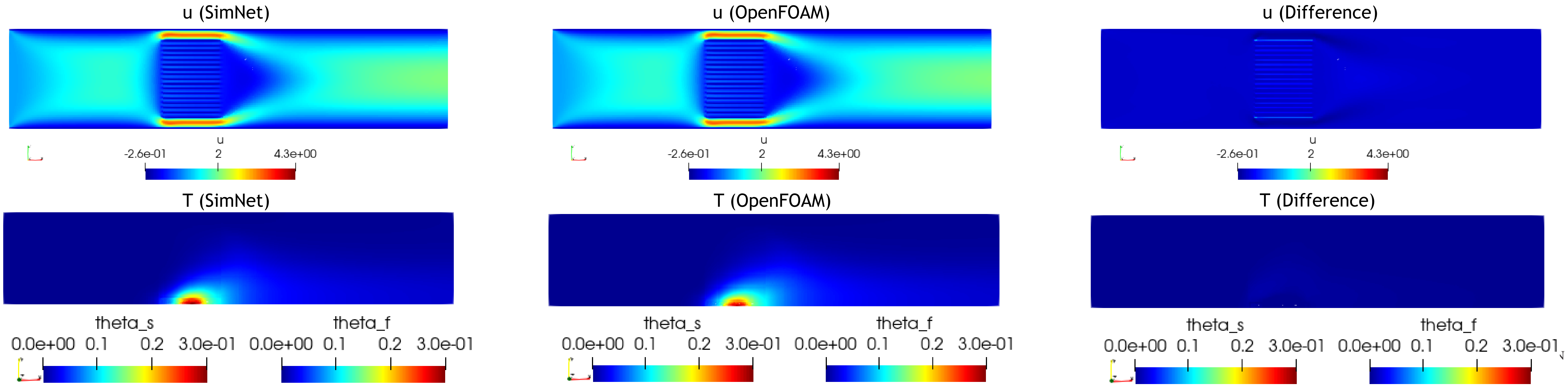
- Thin fin spacing causes sharp gradients.
- Makes it challenging to learn flow inside heatsink.
- **SDF loss weighting** & **IC planes** are used.
- A Zero-Equation turbulence model is used (Re=13k).

FPGA heatsink geometry



Modulus streamlines and temperature



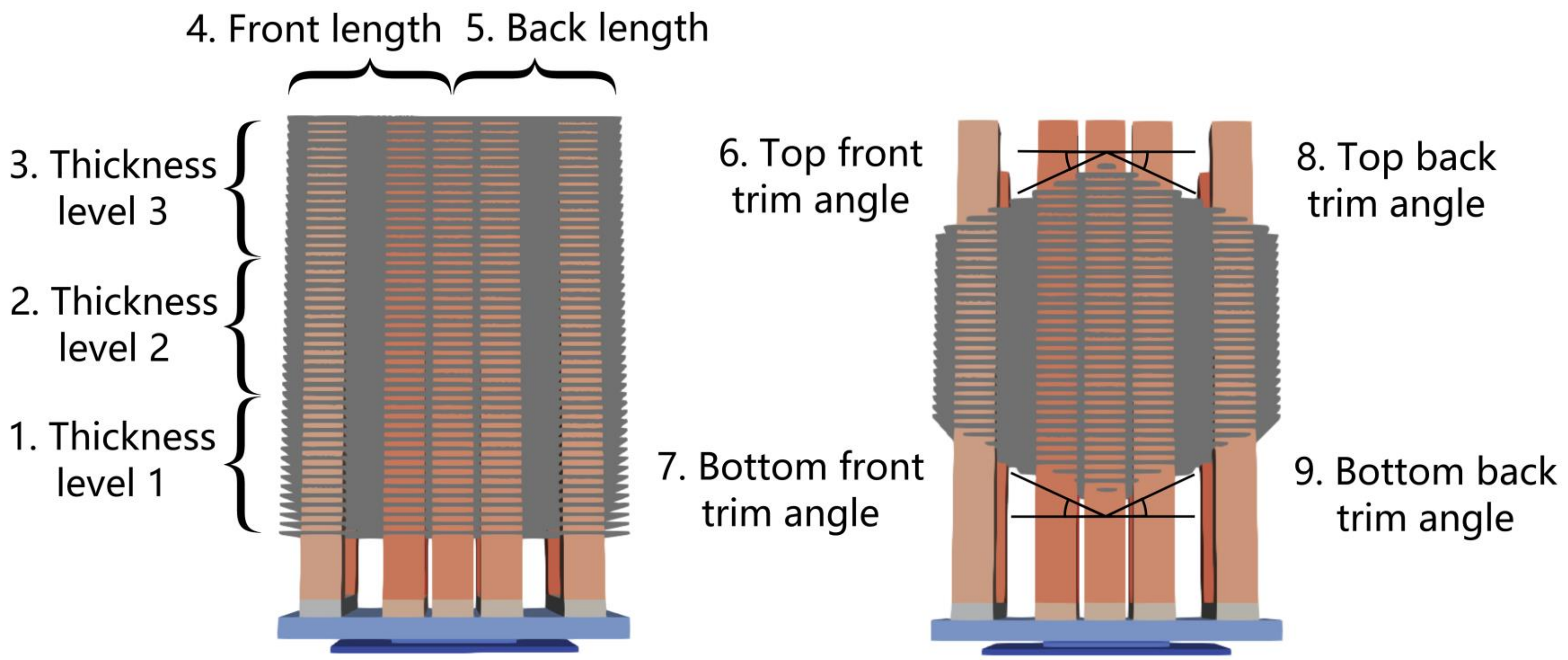


Design optimization for industrial systems

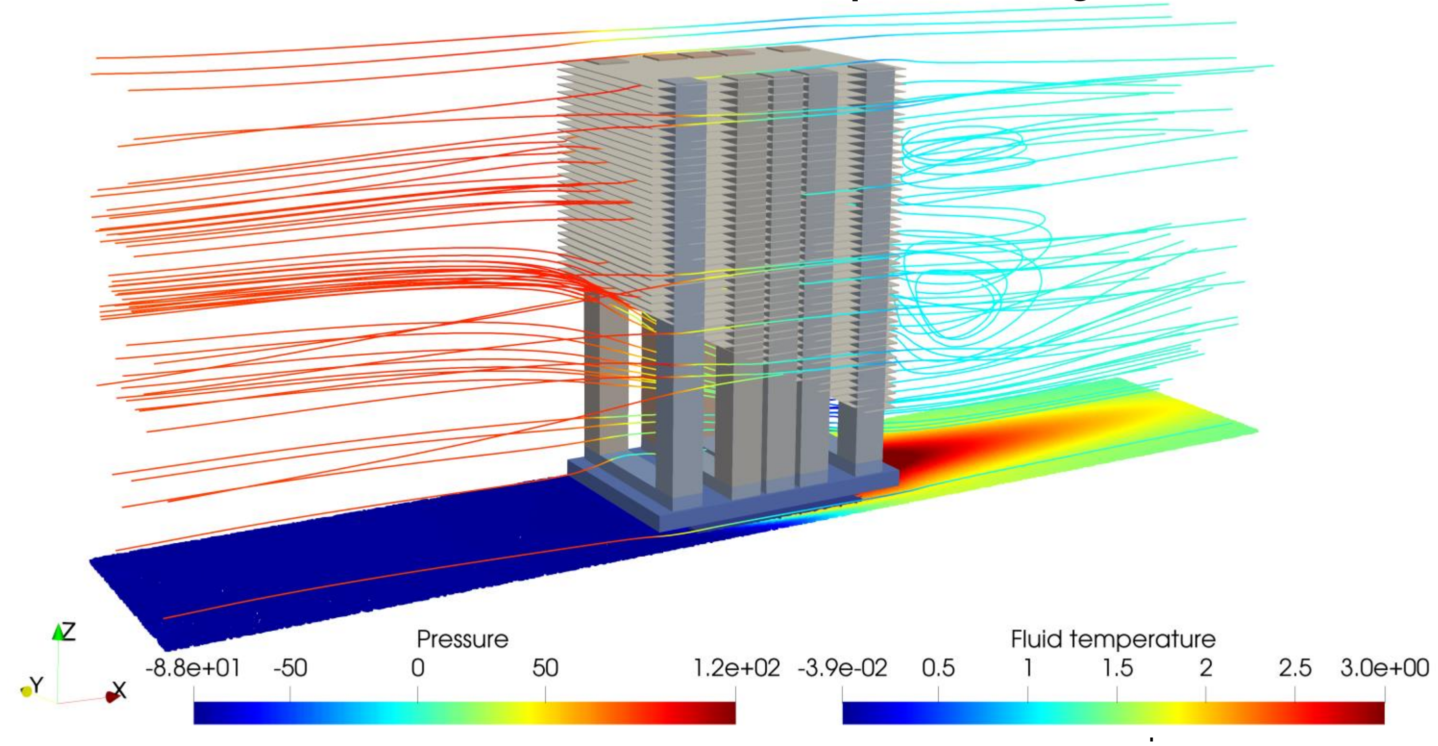
Demonstrating Modulus ability to perform efficient design space exploration.

- Modulus solves several simultaneous design configurations much more efficiently than traditional solvers.
- Unlike a traditional solver, a neural network trains with multiple design parameters in a single training run.
- Once training is complete, several parameter combinations can be evaluated using inference as a post-processing step.
- Here, we train a conjugate heat transfer problem over the Nvidia's NVSwitch heat sink with 9 fin geometry variables.
- By parameterizing geometry, Modulus accelerates design optimization by orders of magnitude vs. traditional solvers.

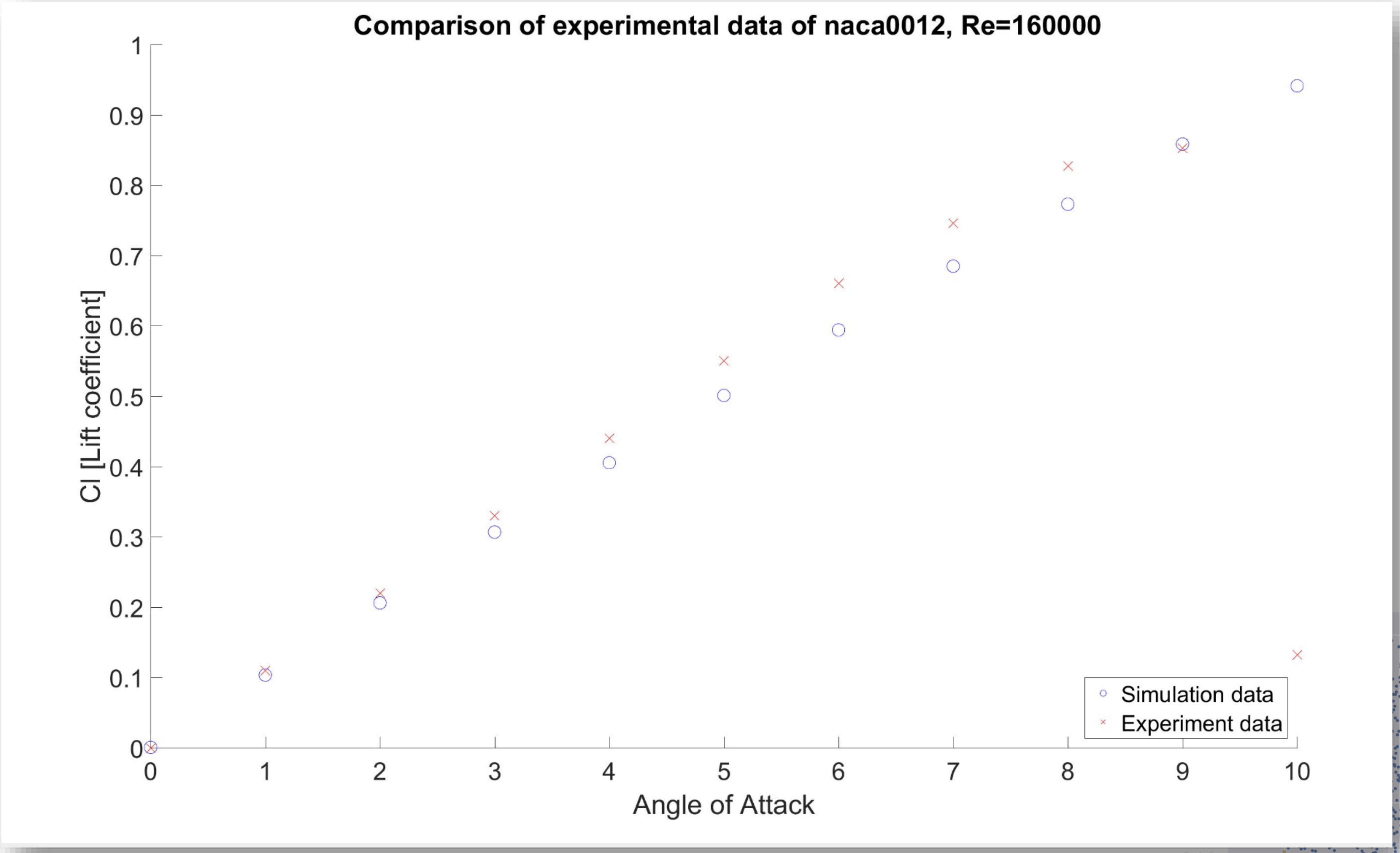
Design parameters



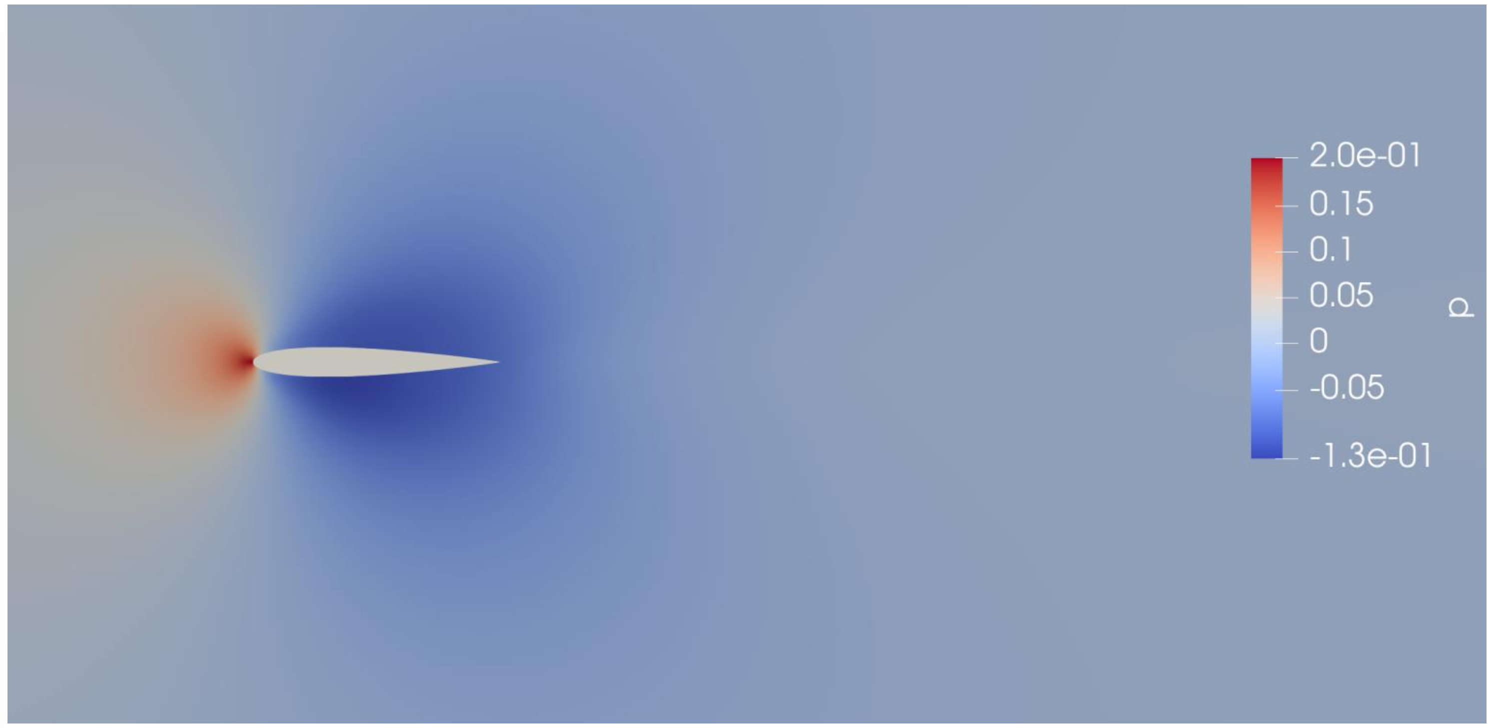
Modulus solution for the optimal design

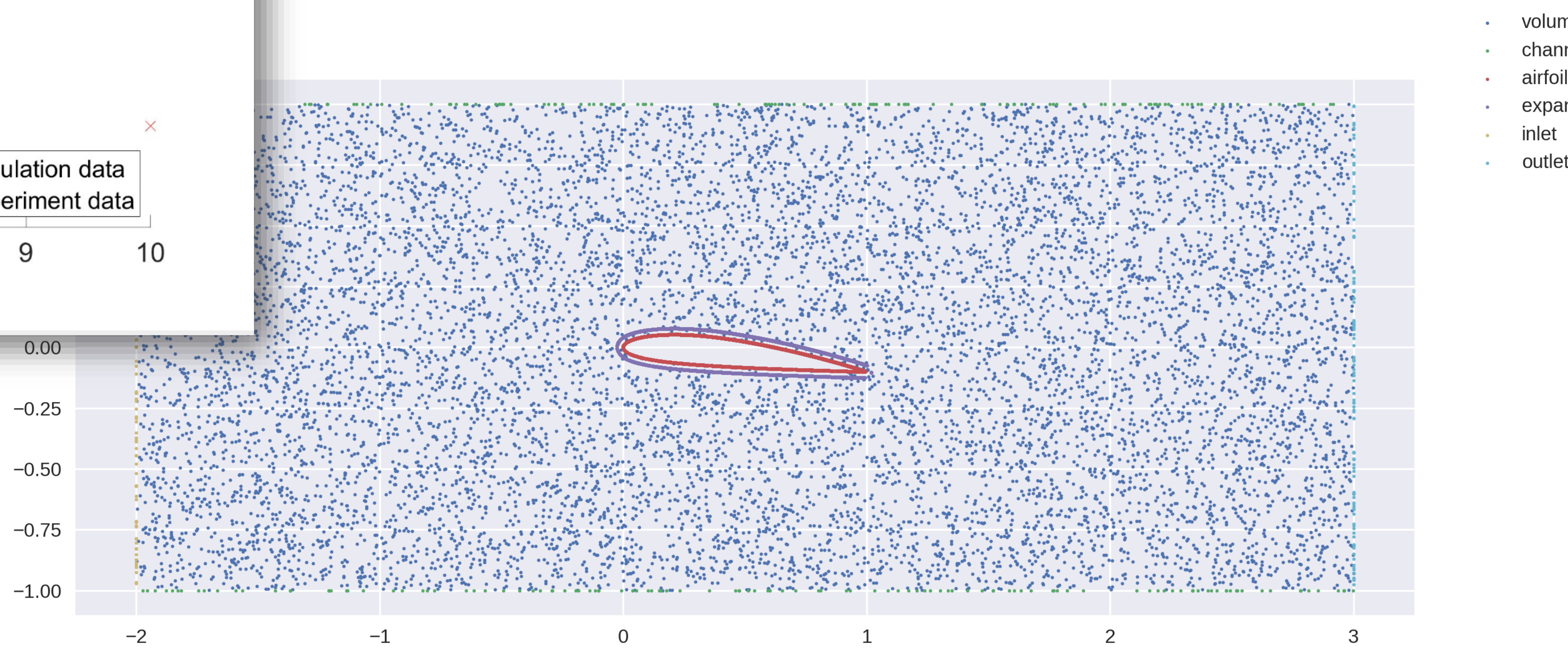


2D Virtual Wind Tunnel



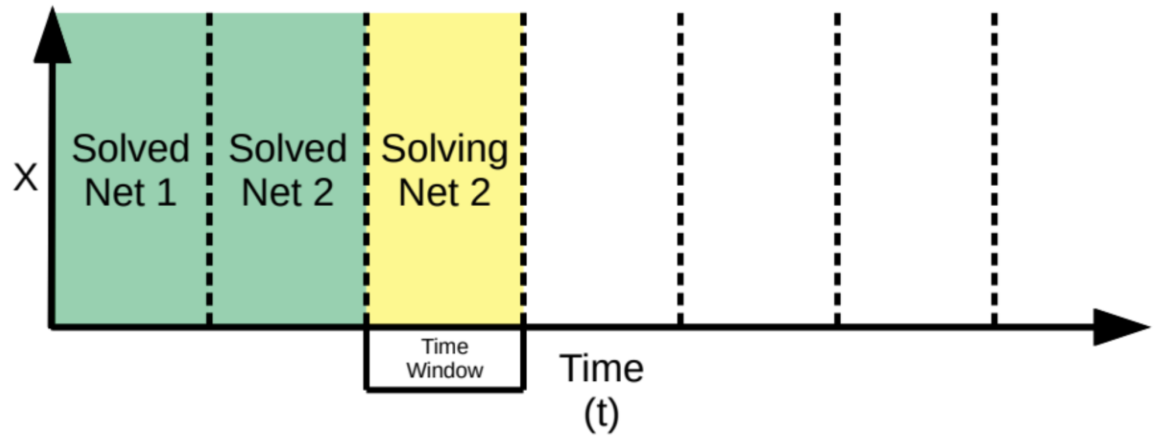
From: https://quickersim.com/validation-of-naca0012-airfoil-for-moderate-values-of-reynolds-numbers/

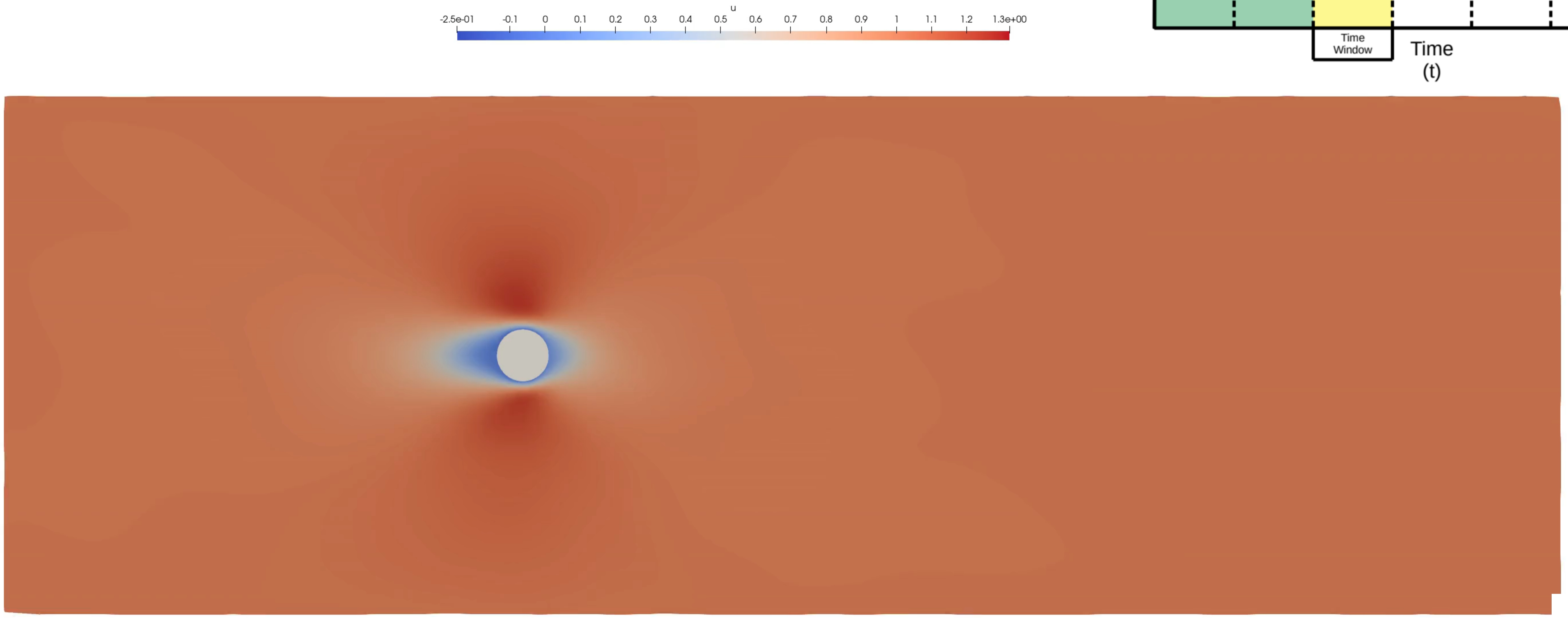




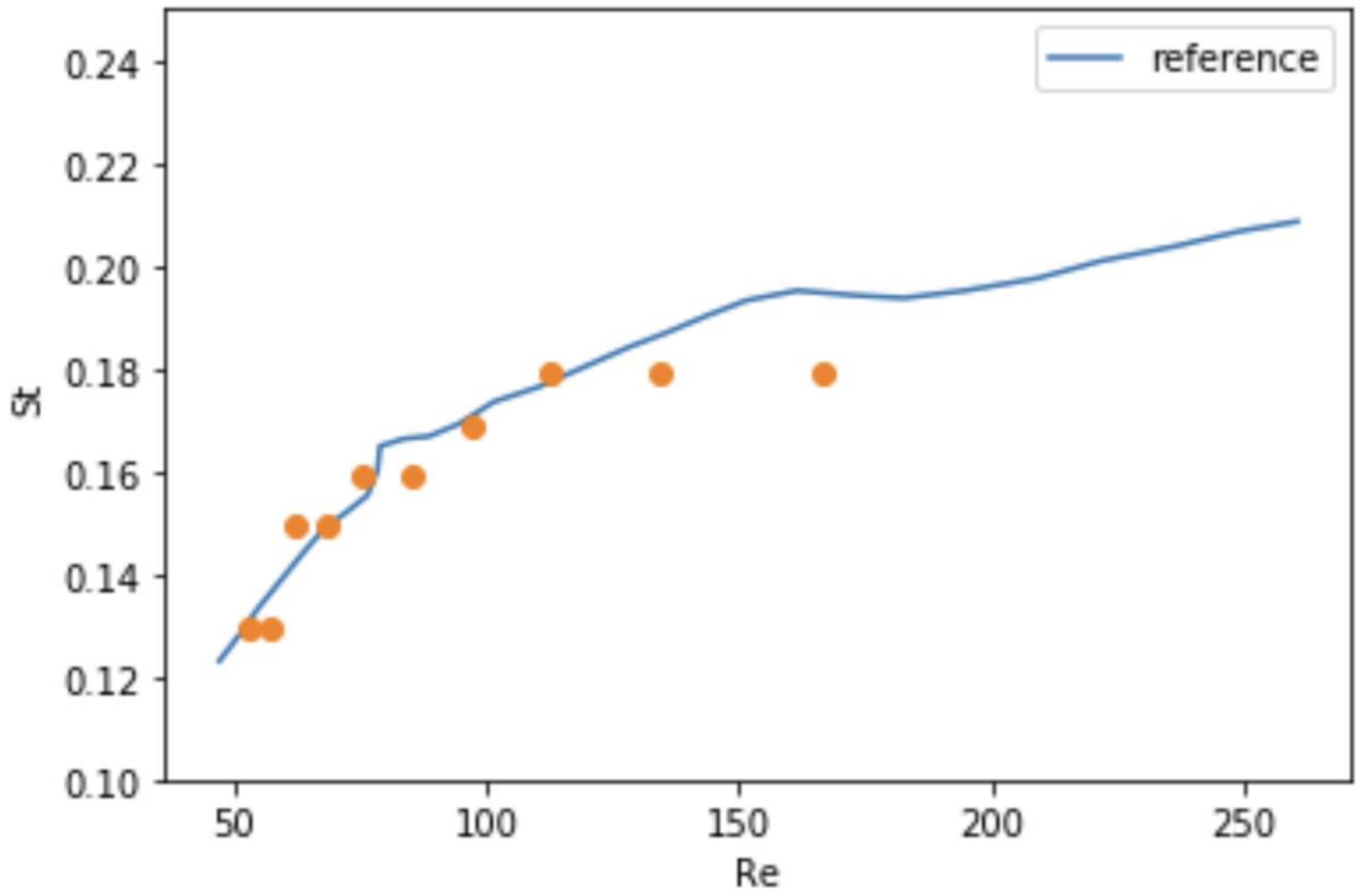
Unsteady problems

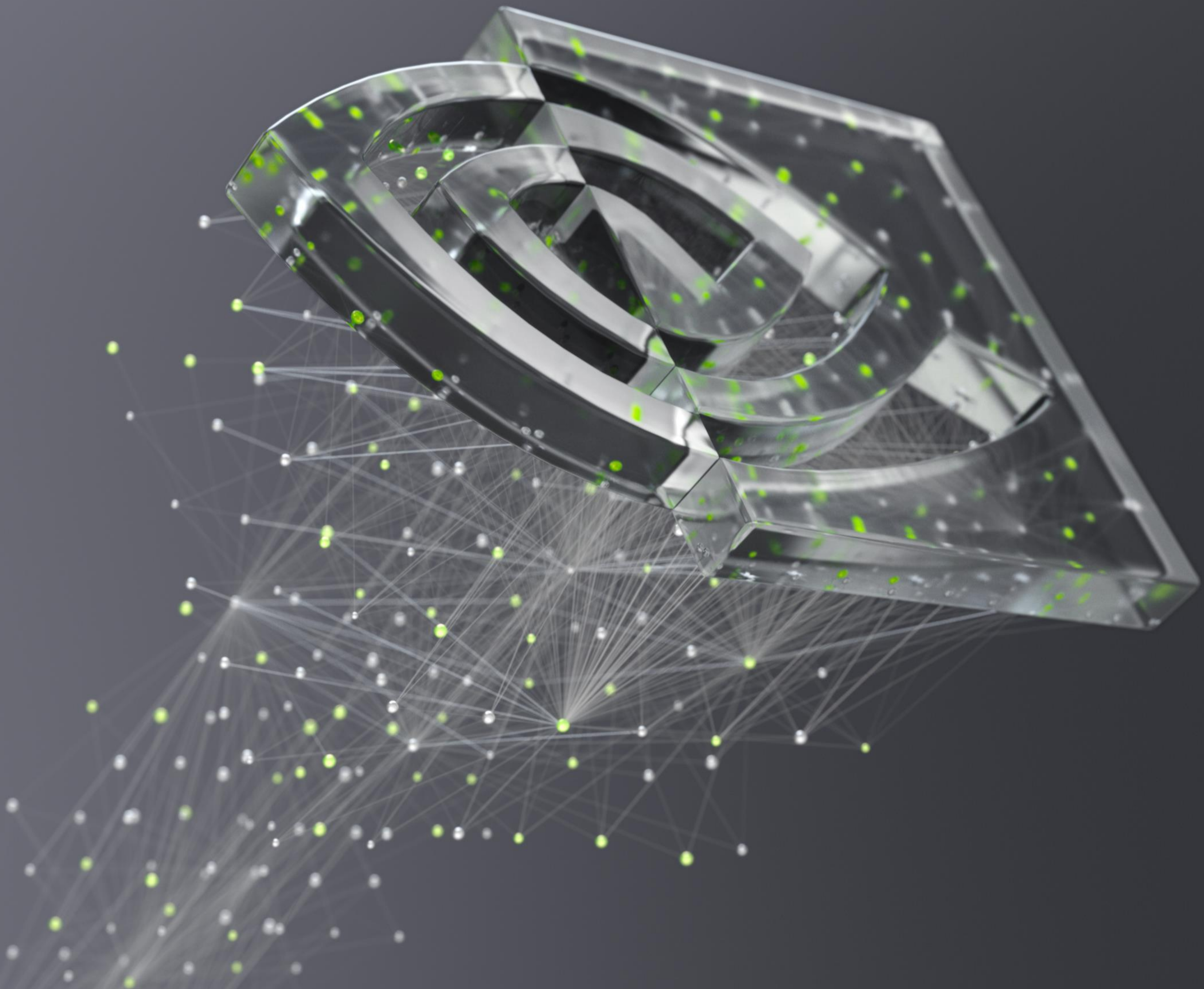
Flow over a 2D cylinder with a parameterised Re





Unsteady problems



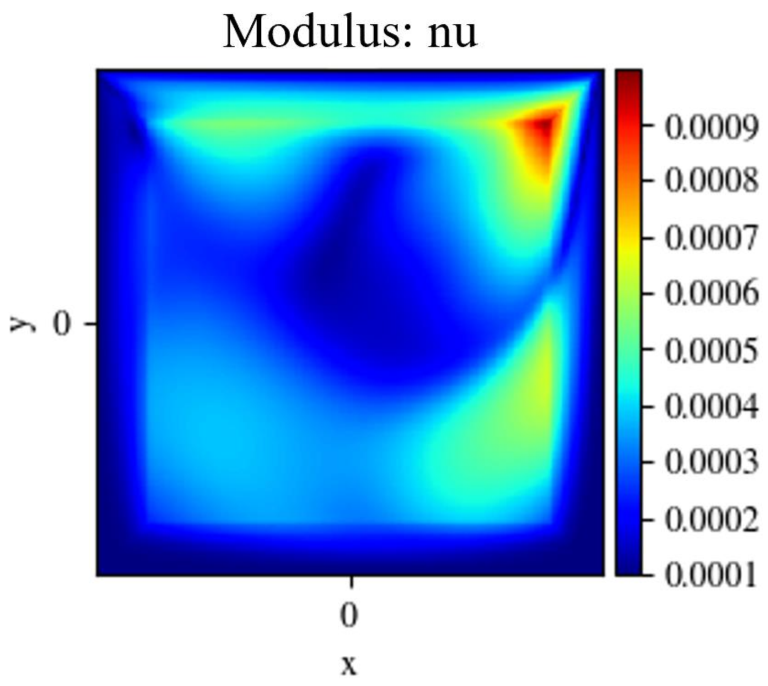


NVIDIA

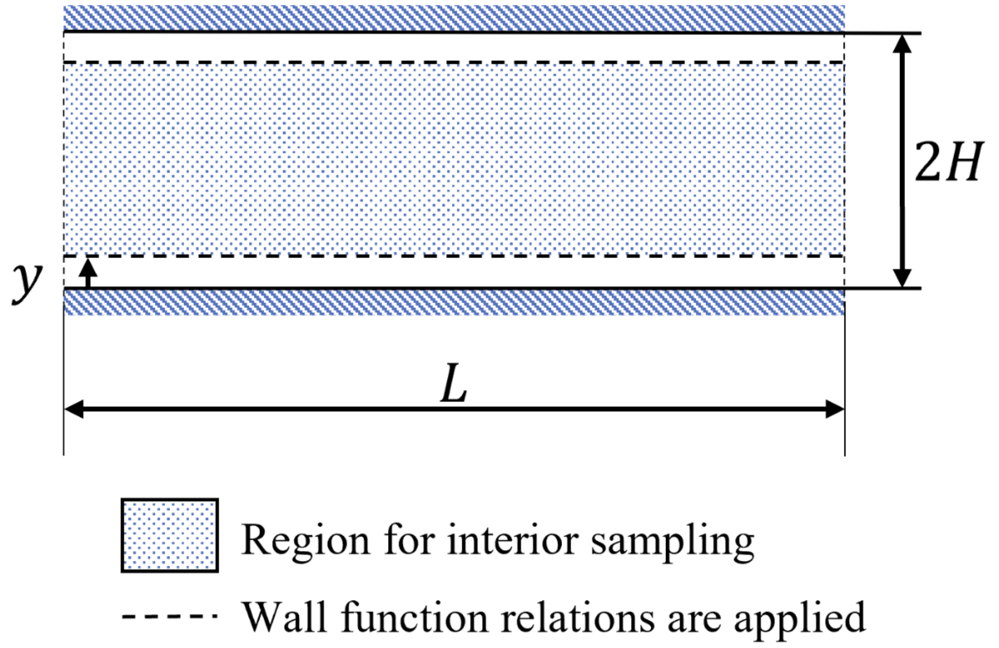
TOWARDS HIGH-RE APPLICATIONS

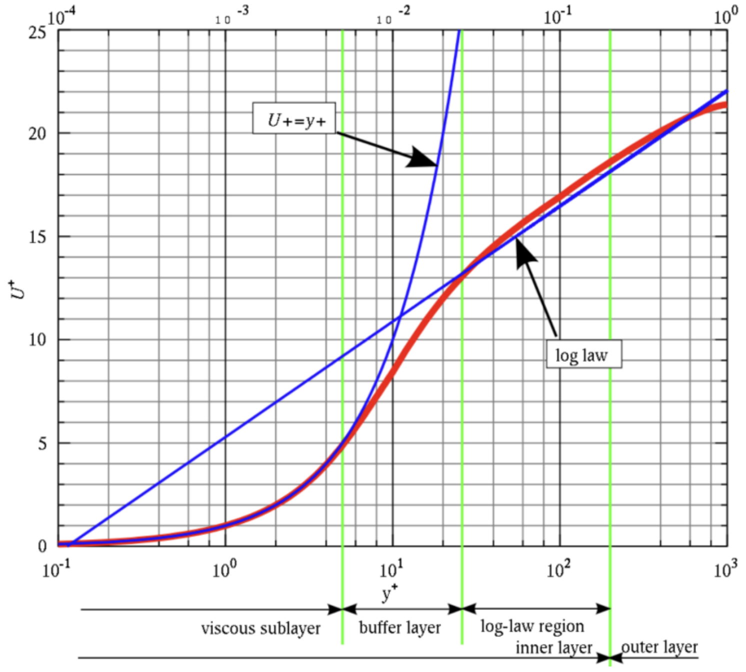
Turbulence modelling

- 0-eq mixing length model



- 2-eq models with wall-functions





TOWARDS HIGH-RE APPLICATIONS

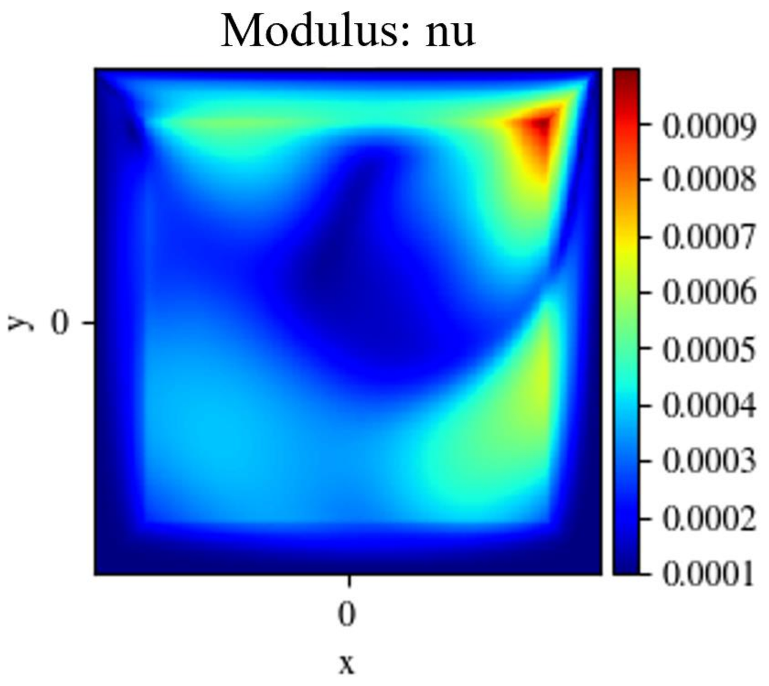
Turbulence modelling

- 0-eq mixing length model

$$\mu_t = \rho l_m^2 \sqrt{G}$$

$$G = 2(u_x)^2 + 2(v_y)^2 + 2(w_z)^2 + (u_y + v_x)^2 + (u_z + w_x)^2 + (v_z + w_y)^2$$

$$l_m = \min(0.419d, 0.09d_{max})$$



- 2-eq models with wall-functions

$$\frac{\partial k}{\partial t} + U \cdot \nabla k = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_k} \right) \nabla k \right] + P_k - \varepsilon$$

$$\frac{\partial \varepsilon}{\partial t} + U \cdot \nabla \varepsilon = \nabla \cdot \left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon} \right) \nabla \varepsilon \right] + (C_{\varepsilon 1} P_k - C_{\varepsilon 2}$$

$$U = \frac{u_\tau}{\kappa} \ln \left(E y^+ \right) \qquad k = \frac{u_\tau^2}{\sqrt{C_\mu}} \qquad \varepsilon = \frac{C_\mu^{3/4} k^{3/2}}{\kappa y}$$