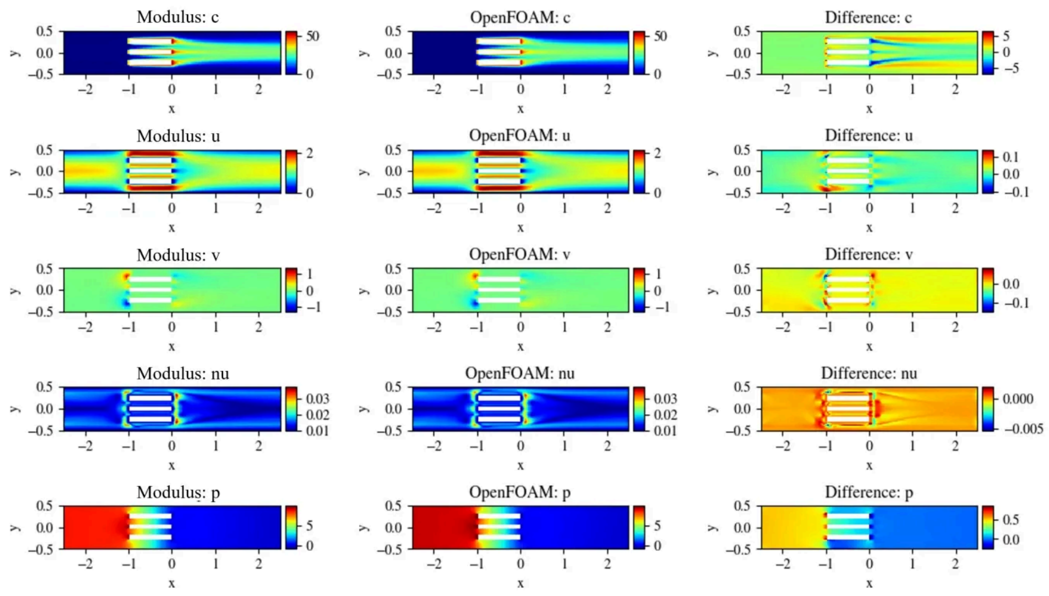
NVIDIA Modulus Part 4: Scalar Transport: 2D Advection Diffusion (Heat dissipation)



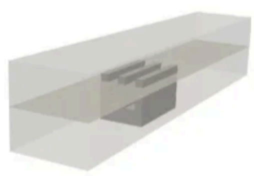
https://docs.nvidia.com/deeplearning/modulus/modulus-sym/user_guide/foundational/scalar_transport.html

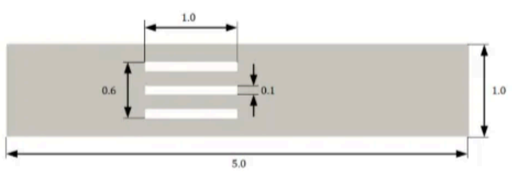
https://catalog.ngc.nvidia.com/orgs/nvidia/teams/modulus/resources/modulus_sym_examples_supplemental_materials

https://github.com/AI-ME-Ben/NVIDIA-Modulus-Reproduce/tree/main

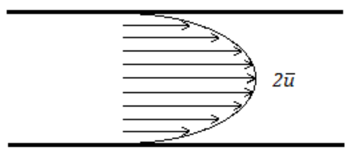
Part 1 : Problem description

In this tutorial, you will solve the heat transfer from a 3-fin heat sink. The problem describes a hypothetical scenario wherein a 2D slice of the heat sink is simulated as shown in the figure. The heat sinks are maintained at a constant temperature of 350 K and the inlet is at 293.498 K. The channel walls are treated as adiabatic. The inlet is assumed to be a parabolic velocity profile with 1.5 m/s as the peak velocity. The kinematic viscosity ν is set to 0.01 m^2/s and the Prandtl number is 5. Although the flow is laminar, the Zero Equation turbulence model is kept on.





Parabolic velocity



Constant temperature: heat sink boundary

Adiabatic: top, bottom

Part 2 : Physics equation

Equilibrium Equation:

Navier Stokes Equation → Flow

$$
\begin{aligned}
\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= 0 \\
u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\
u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)
\end{aligned}
$$

Zero Equation → adjust Navier Stokes Equation

https://docs.nvidia.com/deeplearning/modulus/modulus-sym/user_guide/foundational/zero_eq_turbulence.html

2. Modified Navier-Stokes Equations with Effective Viscosity

In turbulence models, we replace **molecular viscosity** ν with **effective viscosity** ν_{eff}:

$$\nu_{\text{eff}} = \nu + \nu_t$$

where:

- ν = molecular kinematic viscosity
- ν_t = turbulent viscosity (depends on turbulence model)

Now, the modified equations become:

$$
\begin{aligned}
u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= -\frac{\partial p}{\partial x} + \nu_{\text{eff}}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\
u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= -\frac{\partial p}{\partial y} + \nu_{\text{eff}}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)
\end{aligned}
$$

where ν_{eff} accounts for both **laminar and turbulent** diffusion effects.

1. Definition of Effective Viscosity

In the `ZeroEquation` class, the **effective viscosity** ν_{eff} is computed as:

$$\nu_{\text{eff}} = \nu + \rho \cdot l_m^2 \cdot \sqrt{G}$$

where:

- ν = **kinematic viscosity** (molecular viscosity of the fluid)
- ρ = **density** of the fluid
- l_m = **mixing length** (describes turbulence scale, dependent on wall distance)
- G = **strain rate magnitude**, which represents turbulence production

Advection diffusion equation → Heat

$$c = (T_{actual} - T_{inlet})/273.15$$

$$\frac{\partial C}{\partial t} + \mathbf{u} \cdot \nabla C = \nabla \cdot (D \nabla C)$$

evolution term, advection term, diffusion term

$$uc_x + vc_y = D(c_{xx} + c_{yy})$$

https://www.youtube.com/watch?v=YiIT3p507S0

Part 3 : Code walkthrough

1. **Geometry**

```python
# params for domain
channel_length = (-2.5, 2.5)
channel_width = (-0.5, 0.5)
heat_sink_origin = (-1, -0.3)
nr_heat_sink_fins = 3
gap = 0.15 + 0.1
heat_sink_length = 1.0
heat_sink_fin_thickness = 0.1
inlet_vel = 1.5
heat_sink_temp = 350
base_temp = 293.498
nu = 0.01
diffusivity = 0.01 / 5
```

```python
# define sympy varaibles to parametize domain curves
x, y = Symbol("x"), Symbol("y")

# define geometry
channel = Channel2D(
    (channel_length[0], channel_width[0]), (channel_length[1], channel_width[1])
)
heat_sink = Rectangle(
    heat_sink_origin,
    (
        heat_sink_origin[0] + heat_sink_length,
        heat_sink_origin[1] + heat_sink_fin_thickness,
    ),
)
for i in range(1, nr_heat_sink_fins):
    heat_sink_origin = (heat_sink_origin[0], heat_sink_origin[1] + gap)
    fin = Rectangle(
        heat_sink_origin,
        (
            heat_sink_origin[0] + heat_sink_length,
            heat_sink_origin[1] + heat_sink_fin_thickness,
        ),
    )
    heat_sink = heat_sink + fin
geo = channel - heat_sink
```

```python
inlet = Line(
    (channel_length[0], channel_width[0]), (channel_length[0], channel_width[1]), -1
)
outlet = Line(
    (channel_length[1], channel_width[0]), (channel_length[1], channel_width[1]), 1
)
```

2. **Neural networks nodes**

```python
    # make list of nodes to unroll graph on
    ze = ZeroEquation(
        nu=nu, rho=1.0, dim=2, max_distance=(channel_width[1] - channel_width[0]) / 2
    )
    ns = NavierStokes(nu=ze.equations["nu"], rho=1.0, dim=2, time=False)
    ade = AdvectionDiffusion(T="c", rho=1.0, D=diffusivity, dim=2, time=False)
    gn_c = GradNormal("c", dim=2, time=False)
    normal_dot_vel = NormalDotVec(["u", "v"])
    flow_net = instantiate_arch(
        input_keys=[Key("x"), Key("y")],
        output_keys=[Key("u"), Key("v"), Key("p")],
        cfg=cfg.arch.fully_connected,
    )
    heat_net = instantiate_arch(
        input_keys=[Key("x"), Key("y")],
        output_keys=[Key("c")],
        cfg=cfg.arch.fully_connected,
    )
```

```python
nodes = (
    ns.make_nodes()
    + ze.make_nodes()
    + ade.make_nodes(detach_names=["u", "v"])
    + gn_c.make_nodes()
    + normal_dot_vel.make_nodes()
    + [flow_net.make_node(name="flow_network")]
    + [heat_net.make_node(name="heat_network")]
)
```

3. Constraint

```python
    # make domain
    domain = Domain()

    # inlet
    inlet_parabola = parabola(
        y, inter_1=channel_width[0], inter_2=channel_width[1], height=inlet_vel
    )
    inlet = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=inlet,
        outvar={"u": inlet_parabola, "v": 0, "c": 0},
        batch_size=cfg.batch_size.inlet,
    )
    domain.add_constraint(inlet, "inlet")

    # outlet
    outlet = PointwiseBoundaryConstraint(
        nodes=nodes,
        geometry=outlet,
        outvar={"p": 0},
        batch_size=cfg.batch_size.outlet,
    )
    domain.add_constraint(outlet, "outlet")
```

```python
# heat_sink wall
hs_wall = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=heat_sink,
    outvar={"u": 0, "v": 0, "c": (heat_sink_temp - base_temp) / 273.15},
    batch_size=cfg.batch_size.hs_wall,
)
domain.add_constraint(hs_wall, "heat_sink_wall")

# channel wall
channel_wall = PointwiseBoundaryConstraint(
    nodes=nodes,
    geometry=channel,
    outvar={"u": 0, "v": 0, "normal_gradient_c": 0},
    batch_size=cfg.batch_size.channel_wall,
)
domain.add_constraint(channel_wall, "channel_wall")
```

GradNormal → Adiabatic

$$\frac{\partial T}{\partial x} = 0 \qquad \frac{\partial T}{\partial y} = 0$$

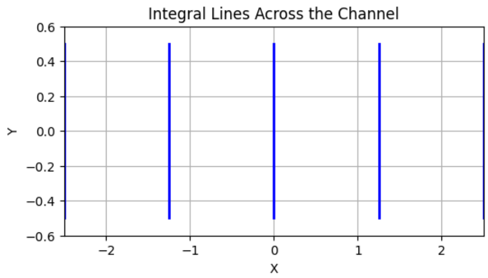
```python
# interior flow
interior_flow = PointwiseInteriorConstraint(
    nodes=nodes,
    geometry=geo,
    outvar={"continuity": 0, "momentum_x": 0, "momentum_y": 0},
    batch_size=cfg.batch_size.interior_flow,
    compute_sdf_derivatives=True,
    lambda_weighting={
        "continuity": Symbol("sdf"),
        "momentum_x": Symbol("sdf"),
        "momentum_y": Symbol("sdf"),
    },
)
domain.add_constraint(interior_flow, "interior_flow")

# interior heat
interior_heat = PointwiseInteriorConstraint(
    nodes=nodes,
    geometry=geo,
    outvar={"advection_diffusion_c": 0},
    batch_size=cfg.batch_size.interior_heat,
    lambda_weighting={
        "advection_diffusion_c": 1.0,
    },
)
domain.add_constraint(interior_heat, "interior_heat")
```

NormalDotVec → integral continuity

added as an additional constraint to speed up the convergence

```
x_pos = Parameter("x_pos")
integral_line = Line(
    (x_pos, channel_width[0]),
    (x_pos, channel_width[1]),
    1,
    parameterization=Parameterization({x_pos: channel_length}),
)
```



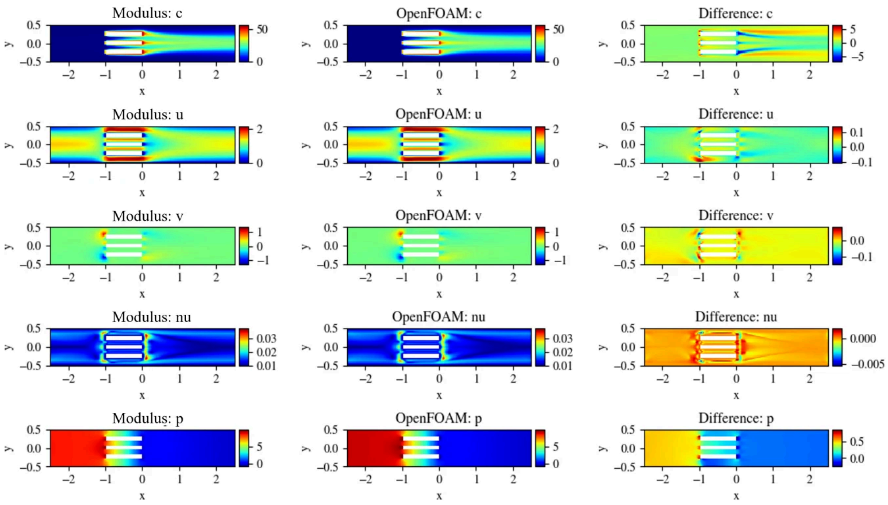

```
# integral continuity
def integral_criteria(invar, params):
    sdf = geo.sdf(invar, params)
    return np.greater(sdf["sdf"], 0)

integral_continuity = IntegralBoundaryConstraint(
    nodes=nodes,
    geometry=integral_line,
    outvar={"normal_dot_vel": 1},
    batch_size=cfg.batch_size.num_integral_continuity,
    integral_batch_size=cfg.batch_size.integral_continuity,
    lambda_weighting={"normal_dot_vel": 0.1},
    criteria=integral_criteria,
)
domain.add_constraint(integral_continuity, "integral_continuity")
```

Part 4: Code Implementation

https://github.com/AI-ME-Ben/NVIDIA-Modulus-Reproduce/tree/main/NVIDIA%20Modulus%20Part%204%20Scalar%20Transport%202D%20Advection%20Diffusion%

```
# add validation data
file_path = "openfoam/heat_sink_zeroEq_Pr5_mesh20.csv"
```



attachment:b1b22bf1-1644-4647-8444-786390268e9c:t.mp4