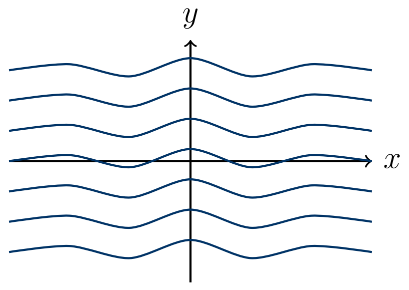
flowINN

Physics-Informed Neural Networks for Fluid Dynamics



A Framework for Solving Navier-Stokes Equations
with Neural Networks on Arbitrary Geometries

FLOW INFORMED NEURAL NETWORKS

flowINN

February 13, 2025

Contents

Chapter 1

Introduction

This document provides an overview of the **fl0wINN** framework, an open-source Python library designed for developing and applying Physics-Informed Neural Networks (PINNs) to solve fluid dynamics problems. PINNs leverage the power of deep learning to directly solve partial differential equations (PDEs) governing fluid flow, such as the Navier-Stokes equations, without the need for traditional mesh-based discretization methods like Finite Element Method (FEM) or Finite Volume Method (FVM).

Key Features:

- **Modular Design:** The framework is structured with clear separation of concerns, encompassing modules for:
  - **Mesh Generation:** Handling mesh creation, including 2D and 3D meshes with various sampling methods (e.g., random, uniform), boundary condition definitions, and interior boundary handling.
  - **Physics:** Encapsulating the physics of the problem, such as governing equations (e.g., Navier-Stokes), boundary conditions, and source terms. Software has been built so that physics is easily replaceable as the project develops.
  - **Models:** Defining the neural network architectures and training procedures for PINNs. As new and more advanced architectures are developed, implementation will be straightforward.
  - **Training:** Implementing loss functions and optimization algorithms for training the PINNs.
  - **Plotting:** Providing utilities for visualizing simulation results, including velocity fields, pressure distributions, and other relevant quantities. For more advanced visualization, it is possible to dump csv files for further postprocessing in Paraview.
- **Flexibility:** The framework is designed to be adaptable to a wide range of fluid dynamics problems, including:
  - Incompressible Flow
  - Compressible Flow
  - Turbulent Flow
  - Multiphase Flow

- **User-Friendly Interface:** The framework offers a user-friendly interface with well-documented APIs, making it accessible for researchers and engineers to implement and experiment with different PINN models.
- **Example Applications:** Includes several example applications demonstrating the use of fl0wINN for solving classic fluid dynamics problems, such as:
  - Flow Over Airfoil
  - Lid-Driven Cavity
  - Channel Flow

Project Structure:

- `src`: Contains the core source code of the framework, organized into modules for meshing, physics, models, training, plotting, and utilities.
- `examples`: Provides Python scripts demonstrating the use of fl0wINN for specific fluid dynamics problems.
- `jupyterNotebooks`: Provides interactive Jupyter Notebooks for exploring and experimenting with the framework, with seamless export to Google Colab.

Getting Started:

1. **Installation:** Install the necessary dependencies using `pip install -r requirements.txt`.
2. **Exploration:** Explore the provided examples and Jupyter Notebooks to gain familiarity with the framework.
3. **Customization:** Modify existing examples or create new ones to solve your specific fluid dynamics problems.
4. **Development:** Contribute to the development of the framework by adding new features, improving existing functionalities, and addressing issues.

fl0wINN aims to be a valuable tool for researchers and engineers interested in leveraging the power of deep learning to advance the field of computational fluid dynamics.

Chapter 2

Steady Incompressible Navier-Stokes equation

The implementation covers the steady-state incompressible Navier-Stokes equations in both 2D and 3D configurations. The governing equations consist of the continuity equation (conservation of mass) and momentum equations.

2.0.1 Continuity Equations

- 2D:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.1}$$

- 3D:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.2}$$

2.0.2 Momentum Equations

2D Configuration

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \tag{2.3}$$

$$u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) \tag{2.4}$$

3D Configuration

Additional z-direction momentum equation:

$$u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) \tag{2.5}$$

2.1 Implementation

2.1.1 NavierStokes2D Class

The `NavierStokes2D` class implements the 2D incompressible Navier-Stokes equations.

Constructor

```python
def __init__(self, nu: float = 0.01) -> None:
    """Initialize with kinematic viscosity (nu)."""
    self.nu = nu
```

Parameters:

- `nu`: Kinematic viscosity (default: 0.01).
  - Must be a positive float value.
  - Implemented with property decorators for validation.

Core Methods

```python
def _compute_first_derivatives(
    self, u, v, p, x, y, tape
) -> tuple:
```

Parameters:

- `u, v`: Velocity components (TensorFlow tensors).
- `p`: Pressure field (TensorFlow tensor).
- `x, y`: Spatial coordinates (TensorFlow tensors).
- `tape`: TensorFlow GradientTape instance.

Returns: Tuple of $(u_x, u_y, v_x, v_y, p_x, p_y)$

2.1.2 NavierStokes3D Class

The `NavierStokes3D` class extends the implementation to three dimensions.

Core Method

```python
def get_residuals(
    self, u, v, w, p, x, y, z, tape
) -> tuple:
```

Parameters:

- `u, v, w`: Velocity components (TensorFlow tensors).
- `p`: Pressure field (TensorFlow tensor).
- `x, y, z`: Spatial coordinates (TensorFlow tensors,
- `tape`: TensorFlow GradientTape instance,

Returns: Tuple of (continuity, momentum_u, momentum_v, momentum_w)

2.2 Implementation Notes

2.2.1 Automatic Differentiation

All derivatives are computed using TensorFlow's automatic differentiation capabilities:

- The `GradientTape` context manager tracks operations.
- Enables efficient computation of complex derivatives.
- Maintains computational graph for backpropagation.
- All derivatives are computed using automatic differentiation, avoiding numerical finite differences, but still subject to floating-point precision limitations.

2.2.2 Error Handling

Comprehensive error handling is implemented:

- Input validation for kinematic viscosity
- Type checking for tensor inputs
- Shape consistency validation
- Gradient computation validation

2.2.3 Usage Example

```python
import tensorflow as tf

# Initialize solver
ns2d = NavierStokes2D(nu=0.01)

# Create input tensors
u = tf.Variable(...)  # Velocity in x-direction
v = tf.Variable(...)  # Velocity in y-direction
p = tf.Variable(...)  # Pressure field
x = tf.Variable(...)  # x-coordinates
y = tf.Variable(...)  # y-coordinates

# Compute residuals
with tf.GradientTape(persistent=True) as tape:
    continuity, momentum_x, momentum_y = \
        ns2d.get_residuals(u, v, p, x, y, tape)
```

2.3 Common Issues and Solutions

2.3.1 Gradient Computation Failures

- Ensure proper tensor connectivity in computation graph
- Use `persistent=True` for multiple gradient computations
- Verify tensor shapes before computation

2.3.2 Numerical Stability

- Normalize input variables when possible
- Monitor derivative magnitudes
- Consider appropriate pressure scaling

2.3.3 Memory Management

- Release tape resources using `del tape`
- Use `tf.device` for computation placement
- Monitor memory usage during large computations

Chapter 3

Loss Function

3.1 Introduction

This chapter provides detailed documentation for the `NavierStokesLoss` class, a Python implementation for solving the Navier-Stokes equations using Physics-Informed Neural Networks (PINNs). The class combines physical constraints with boundary conditions to create a comprehensive loss function for training neural networks.

3.2 Class Overview

The `NavierStokesLoss` class is designed to handle both 2D and 3D Navier-Stokes equations, providing flexibility in computational fluid dynamics simulations.

3.2.1 Class Constructor

```python
def __init__(self, mesh, model, weights=[0.7, 0.3]):
    self._mesh = mesh
    self._model = model
    self._physics_loss = NavierStokes2D() if mesh.is2D
                    else NavierStokes3D()
    self._loss = None
    self._nu = 0.01
```

S

3.2.2 Loss Function Formulation

The total loss function consists of three main components:

$$\mathcal{L}_{\text{total}} = w_p \mathcal{L}_{\text{physics}} + w_b \left(\mathcal{L}_{\text{boundary}} + \mathcal{L}_{\text{interior}}\right) \tag{3.1}$$

Physics Loss

For 2D cases:

$$\mathcal{L}_{\text{physics}} = \frac{1}{N}\sum_{i=1}^{N}\left[\begin{aligned}&\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)^2\\ &+\left(u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+\frac{\partial p}{\partial x}-\nu\nabla^2 u\right)^2\\ &+\left(u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+\frac{\partial p}{\partial y}-\nu\nabla^2 v\right)^2\end{aligned}\right] \tag{3.2}$$

3.3 Implementation Details

3.3.1 Boundary Condition Handling

The class supports various types of boundary conditions:

Dirichlet Boundary Conditions

For a variable ϕ on boundary Γ:

$$\mathcal{L}_{\text{Dirichlet}} = \frac{1}{N_b}\sum_{i=1}^{N_b}(\phi_{\text{pred},i}-\phi_{\text{target},i})^2 \tag{3.3}$$

Neumann Boundary Conditions

For gradient conditions:

$$\mathcal{L}_{\text{Neumann}} = \frac{1}{N_b}\sum_{i=1}^{N_b}\left(\frac{\partial \phi}{\partial n}_{\text{pred},i} - g_{\text{target},i}\right)^2 \tag{3.4}$$

3.3.2 Code Implementation

```python
def compute_physics_loss(self, u_pred, v_pred, p_pred, X, Y, tape):
    continuity, momentum_u, momentum_v = self._physics_loss.
   get_residuals(u_pred, v_pred, p_pred, X, Y, tape)

    f_loss_u = tf.reduce_mean(tf.square(momentum_u))
    f_loss_v = tf.reduce_mean(tf.square(momentum_v))
    continuity_loss = tf.reduce_mean(tf.square(continuity))

    return f_loss_u + f_loss_v + continuity_loss
```

3.4 Usage Examples

3.4.1 Basic Usage

```python
# Initialize mesh and model
mesh = CustomMesh(...)  # User-defined mesh
model = NeuralNetwork(...)  # User-defined neural network

# Create loss function
ns_loss = NavierStokesLoss(mesh, model)

# Set viscosity
ns_loss.nu = 0.01

# Compute loss
loss_value = ns_loss.loss_function()
```

3.4.2 Advanced Configuration

```python
# Custom weights for physics and boundary losses
ns_loss = NavierStokesLoss(
    mesh,
    model,
    weights=[0.8, 0.2]  # Emphasize physics loss
)
```

3.5 Error Handling

The class implements robust error handling for boundary conditions:

```python
try:
    # Process boundary conditions
    boundary_loss = process_boundary(...)
except Exception as e:
    print(f"Warning: Error processing boundary: {str(e)}")
    continue
```

3.6 Performance Considerations

3.6.1 Computational Efficiency

The implementation uses TensorFlow's automatic differentiation for computing derivatives:

- Persistent gradient tapes are used to compute multiple derivatives
- Boundary conditions are processed in batches
- Vector operations are preferred over loops

3.6.2 Memory Management

Important considerations for memory efficiency:

- Gradient tapes are used within context managers

- Temporary tensors are properly managed
- Batch processing is used for large datasets

3.7 Extensions and Future Work

3.7.1 Possible Extensions

- Implementation of turbulence models
- Addition of thermal effects
- Support for compressible flows

3.7.2 Limitations

Current limitations of the implementation:

- Limited to incompressible flows
- Assumes constant fluid properties
- No built-in turbulence modeling

Chapter 4

Mesh Generation and Management

4.1 Overview

The `Mesh` class is a cornerstone of the fl0wINN package, providing robust functionality for generating and managing computational meshes essential for fluid dynamics simulations. It supports both 2D and 3D mesh generation, offering a choice of sampling methods, comprehensive boundary condition handling, and utilities for mesh visualization and export.

4.2 Class Description

4.2.1 Class Definition

```
class Mesh:
    def __init__(self, is2D: bool = True) -> None:
```

4.2.2 Attributes

- `_x, _y, _z`: NumPy arrays storing the x, y, and z (for 3D) coordinates of the mesh points. These are 'None' before mesh generation.
- `_solutions`: A dictionary where keys are strings representing solution field names (e.g., "pressure", "velocity"), and values are NumPy arrays storing the corresponding solution data at each mesh point.
- `_boundaries`: A dictionary storing boundary condition data. Keys are strings representing boundary names (e.g., "inlet", "wall"). Values are dictionaries containing:
  - `x, y, (z)`: NumPy arrays of boundary point coordinates.
  - `[variable_name]`: NumPy arrays of boundary condition values for each variable.
  - `[variable_name_type]`: Strings specifying the type of boundary condition (e.g., "dirichlet", "neumann").
  - `isInterior`: Boolean indicating if the boundary is an interior boundary.
- `_interiorBoundaries`: Similar to `_boundaries`, but stores data for interior boundaries (holes or excluded regions within the domain).

- `_is2D`: A boolean flag indicating whether the mesh is 2D (`True`) or 3D (`False`).
- `meshio`: An instance of the `MeshIO` class, used for mesh input/output operations (e.g., writing to Tecplot files). Initialized lazily.

4.3 Key Functionalities

4.3.1 Mesh Generation

The `generateMesh` method is used to create the mesh points within the defined domain. It supports two primary sampling strategies:

- **Random Sampling:** Points are randomly distributed within the domain, respecting the defined exterior and interior boundaries. This method is generally faster but may result in less uniform point distribution.
- **Uniform Sampling:** Points are placed on a structured grid within the domain. This method provides a more uniform distribution but can be computationally more expensive, especially for complex geometries.

```python
def generateMesh(self, Nx: int = 100, Ny: int = 100,
                 Nz: Optional[int] = None,
                 sampling_method: str = 'random') -> None:
```

- `Nx, Ny, Nz`: Integers specifying the approximate number of mesh points in the x, y, and z (for 3D) directions, respectively. The actual number of points may vary, especially with random sampling and complex boundaries.
- `sampling_method`: A string specifying the sampling method, either 'random' or 'uniform'.

4.3.2 Boundary Condition Handling

The `Mesh` class provides methods for defining both exterior and interior boundary conditions. Boundary conditions are crucial for specifying the behavior of the simulated fluid at the domain boundaries.

```python
def setBoundaryCondition(self, xCoord: np.ndarray,
                         yCoord: np.ndarray,
                         value: np.ndarray,
                         varName: str,
                         boundaryName: str,
                         zCoord: Optional[np.ndarray] = None,
                         interior: bool = False,
                         bc_type: Optional[str] = None) -> None:
```

- `xCoord, yCoord, zCoord`: NumPy arrays of boundary point coordinates.
- `value`: NumPy array of boundary condition values for the specified variable.
- `varName`: String specifying the name of the variable (e.g., "temperature", "pressure").

- `boundaryName`: String identifier for the boundary (e.g., "inlet", "wall").
- `interior`: Boolean flag indicating if the boundary is an interior boundary.
- `bc_type`: String specifying the type of boundary condition (e.g., "dirichlet", "neumann").

4.4 Implementation Details

4.4.1 Mesh Generation Algorithm

The mesh generation process generally involves the following steps:

1. **Boundary Data Validation:** Checks if the provided boundary data is valid (e.g., consistent dimensions, no NaN values).
2. **Boundary Coordinate Processing:** Concatenates boundary coordinates from all defined boundaries.
3. **Interior Point Generation:** Generates points within the domain using the specified sampling method (random or uniform).
4. **Point Filtering (Interior Boundaries):** Removes points that fall inside any defined interior boundaries. This ensures that the mesh excludes the regions defined by interior boundaries.
5. **Mesh Structuring:** Reshapes the resulting point coordinates into the appropriate dimensions based on `Nx`, `Ny`, and `Nz`. The coordinates are stored in the `_x`, `_y`, and `_z` attributes.

4.4.2 Point Sampling Methods

Random Sampling

The random sampling method employs the following procedure:

1. Generates a large number of random points within the bounding box of the domain.
2. Uses Delaunay triangulation of the exterior boundary to efficiently determine which points are inside the domain.
3. Filters out points that are inside interior boundaries (if any are defined) using Delaunay triangulations of the interior boundaries.
4. Randomly selects the required number of points from the remaining valid points.
5. Reshapes the selected points into the mesh structure.

Uniform Sampling

The uniform sampling method operates as follows:

1. Creates a regular grid of points within the domain's bounding box, based on the specified `Nx`, `Ny`, and `Nz`.
2. Uses Delaunay triangulation of the exterior boundary to efficiently determine which grid points are inside the domain.
3. Filters out points that are inside interior boundaries (if any are defined) using Delaunay triangulations of the interior boundaries.
4. The remaining points form the uniform mesh.

4.5 Usage Examples

4.5.1 Basic 2D Mesh Generation

```python
mesh = Mesh(is2D=True)

# Set boundary conditions (example)
x_boundary = np.array([0, 1, 1, 0])
y_boundary = np.array([0, 0, 1, 1])
mesh.setBoundaryCondition(x_boundary, y_boundary, np.array([0, 0, 0, 0])
    , "u", "wall", bc_type="dirichlet") # Example Dirichlet BC


# Generate mesh
mesh.generateMesh(Nx=100, Ny=100, sampling_method='random')

# Visualize the mesh
mesh.showMesh()
```

4.5.2 3D Mesh with Interior Boundaries

```python
mesh = Mesh(is2D=False)

# Set exterior boundary (example)
# ... (define x_outer, y_outer, z_outer, outer_values)
mesh.setBoundaryCondition(x_outer, y_outer, z_outer, outer_values, "u",
    "outer", bc_type="dirichlet")


# Set interior boundary (example)
# ... (define x_inner, y_inner, z_inner, inner_values)
mesh.setBoundaryCondition(x_inner, y_inner, z_inner, inner_values, "u",
    "inner", interior=True, bc_type="dirichlet")

# Generate mesh
mesh.generateMesh(Nx=50, Ny=50, Nz=50, sampling_method='uniform')

# Visualize the mesh
mesh.showMesh()
```

4.5.3 MeshIO Class

The `MeshIO` class handles input/output operations for the mesh, such as reading and writing mesh data and solution variables to files.

Class Definition

```
class MeshIO:
    def __init__(self, mesh) -> None:
        ...
```

Attributes

- `mesh`: The `Mesh` object to which this `MeshIO` instance is associated.
- `variables`: A list of strings representing the names of variables to be written to file (e.g., `["X", "Y", "U", "V", "P"]`).

Methods

- `write_tecplot(filename, variables)`:
  - Writes the mesh and solution data to a Tecplot file.
  - `filename`: The name of the Tecplot file to write.
  - `variables`: An optional list of variable names to write. If `None`, the default variables are used.
- `write_solution(filename, variables)`:
  - Writes the solution data to a CSV file.
  - `filename`: Path to the output file.
  - `variables`: An optional list of variable names to write. If `None`, the default variables are used.
- `set_variables(variables)`:
  - Sets the list of variables to be written to file.
  - `variables`: A list of variable names.

Example Usage

```
mesh_io = MeshIO(mesh)  # Assuming 'mesh' is an instance of the Mesh
    class

# Write solution to a Tecplot file
mesh_io.write_tecplot("output.plt")

# Write solution to a CSV file
mesh_io.write_solution("output.csv")
```

Chapter 5

Boundary Conditions Implementation

5.1 Class Hierarchy

The boundary conditions implementation follows an object-oriented design with a clear hierarchy:

$$\begin{tikzcd}
 & \texttt{BoundaryCondition} \arrow[dl] \arrow[dr] & & \\
\texttt{GradientBC} \arrow[d] & & \texttt{DirichletBC} \arrow[dl] \arrow[d] \arrow[dr] & \\
\texttt{OutletBC} & \texttt{WallBC} & \texttt{InletBC} & \texttt{MovingWallBC}
\end{tikzcd}$$

5.2 Base Classes

5.2.1 BoundaryCondition

The abstract base class for all boundary conditions.

```python
class BoundaryCondition(ABC):
    def __init__(self, name: str):
        self.name = name

    @abstractmethod
    def apply(self, x: tf.Tensor, y: tf.Tensor,
              values: Dict[str, Any],
              tape: Optional[tf.GradientTape] = None
             ) -> Dict[str, tf.Tensor]:
        pass
```

Parameters:

- `name`: String identifier for the boundary condition
- `x, y`: Spatial coordinates as TensorFlow tensors
- `values`: Dictionary containing boundary values for variables
- `tape`: Optional TensorFlow GradientTape for automatic differentiation

5.2.2 GradientBC

Implementation of gradient-based boundary conditions.

```
class GradientBC(BoundaryCondition):
    def apply(self, x: tf.Tensor, y: tf.Tensor,
              values: Dict[str, Any],
              tape: Optional[tf.GradientTape] = None
             ) -> Dict[str, tf.Tensor]:
```

Gradient Specifications:

- **Normal Direction:** Specified by `nx` and `ny` components
- **Directional:** Can be specified in 'x' or 'y' direction
- **Value:** Magnitude of the gradient

5.3 Specific Boundary Conditions

5.3.1 DirichletBC

Implementation of Dirichlet boundary conditions with optional gradient constraints.

```
class DirichletBC(BoundaryCondition):
    def apply(self, x: tf.Tensor, y: tf.Tensor,
              values: Dict[str, Any],
              tape: Optional[tf.GradientTape] = None
             ) -> Dict[str, tf.Tensor]:
```

Features:

- Supports direct value specification
- Optional gradient constraints when tape is provided
- Flexible variable handling through dictionary interface

5.3.2 Wall Boundary Conditions

WallBC

Implementation of no-slip wall boundary conditions.

```
class WallBC(DirichletBC):
    # Default values:
    #   u = 0.0 (no-slip)
    #   v = 0.0 (no-slip)
    #   p = specified or None
```

Characteristics:

- Enforces no-slip condition ($u = v = 0$)
- Flexible pressure handling
- Optional gradient constraints

MovingWallBC

Implementation of moving wall boundary conditions.

```python
class MovingWallBC(DirichletBC):
    # Default values:
    #   u = 1.0 (or specified)
    #   v = 0.0
    #   p = specified or None
```

5.3.3 Flow Boundary Conditions

InletBC

Implementation of inlet boundary conditions.

```python
class InletBC(DirichletBC):
    # Default values:
    #   u = 1.0 (or specified)
    #   v = 0.0
    #   p = specified or None
```

OutletBC

Implementation of outlet boundary conditions using gradients.

```python
class OutletBC(GradientBC):
    def apply(self, x: tf.Tensor, y: tf.Tensor,
              values: Dict[str, Any],
              tape: Optional[tf.GradientTape] = None
             ) -> Dict[str, tf.Tensor]:
```

Special Features:

- Gradient-to-value conversion using small coefficient
- Zero gradient for velocity components in x-direction
- Specified or zero pressure value
- Tensor shape consistency checks

5.4 Implementation Notes

5.4.1 Type Hints

The implementation uses Python type hints for better code clarity:

- `Union`: For multiple allowed types
- `Dict`: For dictionary parameters
- `Any`: For flexible value types
- `Optional`: For parameters that can be None
- `Tuple`: For fixed-size collections

5.4.2 Error Handling

- Gradient tape validation
- Tensor shape consistency checks
- Type conversion for numerical values
- Proper initialization of default values

5.4.3 TensorFlow Integration

- Compatible with TensorFlow automatic differentiation
- Proper tensor type conversion
- Shape matching for boundary values
- Gradient computation support

5.5 Usage Example

```python
# Create boundary conditions
wall = WallBC("wall")
inlet = InletBC("inlet")
outlet = OutletBC("outlet")

# Apply boundary conditions
with tf.GradientTape(persistent=True) as tape:
    # Wall boundary
    wall_values = wall.apply(x_wall, y_wall,
                            {'p': {'gradient': 0.0}},
                            tape)

    # Inlet boundary
    inlet_values = inlet.apply(x_inlet, y_inlet,
                              {'u': {'value': 1.0}},
                              tape)

    # Outlet boundary
    outlet_values = outlet.apply(x_outlet, y_outlet,
                                {'p': {'value': 0.0}},
                                tape)
```

Chapter 6

Model class

6.1 Overview

The PINN class implements a Physics-Informed Neural Network model designed for solving partial differential equations (PDEs). This implementation leverages TensorFlow's computational capabilities and provides a flexible architecture for various PDE-solving scenarios.

6.2 Class Definition

```python
class PINN:
    """Physics-Informed Neural Network (PINN) class"""
```

6.3 Constructor

6.3.1 Method Signature

```python
def __init__(self, input_shape: int = 2,
             output_shape: int = 1,
             layers: List[int] = [20, 20, 20],
             activation: str = 'tanh',
             learning_rate: float = 0.01,
             eq: str = 'LidDrivenCavity') -> None
```

6.3.2 Parameters

- `input_shape` (int): Dimensionality of the input space. Default: 2
- `output_shape` (int): Dimensionality of the output space. Default: 1
- `layers` (List[int]): Architecture of hidden layers. Default: [20, 20, 20]
- `activation` (str): Activation function for hidden layers. Default: 'tanh'
- `learning_rate` (float): Initial learning rate for optimization. Default: 0.01
- `eq` (str): Identifier for the equation being solved. Default: 'LidDrivenCavity'

6.4 Core Methods

6.4.1 create_model

Creates the neural network architecture using TensorFlow's Sequential API.

```
def create_model(self, input_shape: int,
                 output_shape: int,
                 layers: List[int],
                 activation: str) -> tf.keras.Sequential
```

Implementation Details

The method constructs a feedforward neural network with:

- An input layer matching the specified input dimensionality
- Multiple hidden layers with specified units and activation functions
- A linear output layer (no activation function)

6.4.2 learning_rate_schedule

Implements an exponential decay schedule for the learning rate.

```
def learning_rate_schedule(self,
    initial_learning_rate: float
) -> tf.keras.optimizers.schedules.ExponentialDecay
```

Schedule Parameters

- Initial learning rate: User-specified value
- Decay steps: 1000
- Decay rate: 0.9

6.4.3 train_step

Executes a single training iteration using automatic differentiation.

```
@tf.function(jit_compile=True)
def train_step(self,
    loss_function: Callable[], tf.Tensor]
) -> tf.Tensor
```

Implementation Notes

- Utilizes TensorFlow's GradientTape for automatic differentiation
- JIT-compiled for improved performance
- Updates model parameters using the Adam optimizer

6.4.4 train

Implements the main training loop with monitoring and saving capabilities.

```python
def train(self,
    loss_function: Callable[[], tf.Tensor],
    epochs: int = 1000,
    print_interval: int = 100,
    autosave_interval: int = 100,
    plot_loss: bool = False) -> None
```

Features

- Real-time loss plotting (optional)
- Automatic model checkpointing
- Progress monitoring with configurable intervals
- Error handling for model saving operations

6.4.5 predict

Generates predictions using the trained model.

```python
def predict(self, X: np.ndarray) -> np.ndarray
```

6.4.6 load

Loads a previously saved model with error handling.

```python
def load(self, model_name: str) -> None
```

6.5 Usage Example

```python
# Initialize PINN for a 2D problem
pinn = PINN(input_shape=2,
            output_shape=1,
            layers=[32, 32, 32],
            activation='tanh',
            learning_rate=0.001)

# Define loss function
def custom_loss():
    # Implementation of physics-informed loss
    pass

# Train the model
pinn.train(loss_function=custom_loss,
           epochs=5000,
           plot_loss=True)
```

6.6 Best Practices

1. Initialize with appropriate layer sizes based on problem complexity
2. Use 'tanh' activation for PDE problems due to its smoothness properties
3. Monitor training progress using the built-in plotting functionality
4. Implement appropriate physics constraints in the loss function
5. Save models regularly using the autosave feature

6.7 Error Handling

The class implements robust error handling for:

- Model saving operations
- Model loading operations
- Invalid input shapes
- Resource allocation issues

6.8 Performance Considerations

- JIT compilation optimizes training performance
- Exponential learning rate decay helps convergence
- Automatic GPU utilization when available
- Memory-efficient gradient computation

Chapter 7

Postprocessing

7.1 Plot Class

7.1.1 Overview

The `Plot` class provides functionality for visualizing solution fields on computational meshes. It supports both 2D and 3D plotting capabilities, including contour plots, streamlines, scatter plots, and slice views for 3D data.

7.1.2 Class Definition

```python
class Plot:
    def __init__(self, mesh: Mesh) -> None:
```

7.1.3 Key Components

Attributes

- `mesh` (Mesh): The computational mesh object
- `postprocessor` (Optional[Postprocess]): Postprocessing utility object

Main Methods

plot(solkey: str, streamlines: bool) Visualizes solution fields using contour plots with optional streamlines.

$$\begin{gathered}\text{Input field } f(x,y) \text{ interpolated onto regular grid} \\ (x_i, y_j) \in [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]\end{gathered} \tag{7.1}$$

scatterPlot(solkey: str) Creates scatter plots of solution fields with boundary visualization.

For 3D cases, the view scaling is normalized using:

$$\text{max\_range} = \max(\Delta x, \Delta y, \Delta z) \tag{7.2}$$

plotSlices(solkey: str, num_points: int, z_cuts: Optional[list]) Generates slice views for 3D solution fields at specified z-positions:

$$z_i = z_{\text{min}} + \alpha_i(z_{\text{max}} - z_{\text{min}}) \tag{7.3}$$

where α_i are the specified cut positions (default: [0.25, 0.5, 0.75]).

7.2 Postprocess Class

7.2.1 Overview

The `Postprocess` class implements methods for computing derived quantities from simulation results, such as velocity magnitude and pressure coefficients.

7.2.2 Class Definition

```python
class Postprocess:
    def __init__(self, plot_obj: 'Plot') -> None:
```

7.2.3 Computational Methods

Velocity Magnitude

Computes the magnitude of velocity vector:
For 2D flows:

$$|\mathbf{v}| = \sqrt{u^2 + v^2} \tag{7.4}$$

For 3D flows:

$$|\mathbf{v}| = \sqrt{u^2 + v^2 + w^2} \tag{7.5}$$

Pressure Coefficient

Calculates the non-dimensional pressure coefficient:

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty V_\infty^2} \tag{7.6}$$

where:

- p is the local pressure
- p_∞ is the freestream pressure
- ρ_∞ is the freestream density
- V_∞ is the freestream velocity

7.3 Implementation Details

7.3.1 Interpolation Methods

The plotting routines use cubic interpolation for field visualization:

$$f(x, y) = \sum_{i=1}^{n} w_i f_i \tag{7.7}$$

where w_i are the interpolation weights determined by the cubic method.

7.3.2 Boundary Treatment

Boundaries are handled separately for:

- External boundaries (black lines)
- Internal boundaries (red lines)
- Different boundary condition types (Dirichlet, Neumann)

7.3.3 Grid Generation

For visualization, structured grids are generated with:

$$\begin{aligned} x_i &= x_{\min} + i\Delta x, \quad i = 0, \ldots, n_x \\ y_j &= y_{\min} + j\Delta y, \quad j = 0, \ldots, n_y \\ z_k &= z_{\min} + k\Delta z, \quad k = 0, \ldots, n_z \text{ (3D only)} \end{aligned} \tag{7.8}$$

7.4 Usage Examples

7.4.1 Basic Plotting

```python
# Create plot object
plot = Plot(mesh)

# Plot velocity field with streamlines
plot.plot(solkey='u', streamlines=True)
```

7.4.2 Post-processing

```python
# Create postprocessor
post = Postprocess(plot)

# Compute velocity magnitude
post.compute_velocity_magnitude()

# Calculate pressure coefficient
cp = post.compute_pressure_coefficient(
    rho_inf=1.0,
    v_inf=1.0
)
```

7.5 Error Handling

7.5.1 Input Validation

Both classes implement robust error checking:

- Type checking for input parameters
- Verification of solution field existence
- Validation of mesh dimensionality
- Boundary data consistency checks

7.5.2 Common Exceptions

- TypeError: Invalid input types
- KeyError: Missing solution fields
- ValueError: Invalid mesh dimensions

7.6 Performance Considerations

7.6.1 Memory Management

Important considerations for efficient operation:

- Efficient grid interpolation methods
- Optimized numpy operations
- Proper memory cleanup for large datasets

7.6.2 Computational Efficiency

The implementation uses:

- Vectorized operations where possible
- Efficient interpolation algorithms
- Optimized plotting routines

Chapter 8

Example: Flow Over Airfoil

8.1 Overview

The `FlowOverAirfoil` class implements a computational fluid dynamics simulation of flow around a NACA 4-digit airfoil using Physics-Informed Neural Networks (PINN). This class handles mesh generation, boundary conditions, training, and visualization of the flow field.

8.2 Class Definition

```python
class FlowOverAirfoil:
    """Implementation of flow over NACA airfoil using PINN"""
```

8.3 Constructor

8.3.1 Method Signature

```python
def __init__(self, caseName: str,
             xRange: Tuple[float, float],
             yRange: Tuple[float, float],
             AoA: float = 0.0)
```

8.3.2 Parameters

- `caseName` (str): Unique identifier for the simulation
- `xRange` (Tuple[float, float]): Domain extent in x-direction (min_x, max_x)
- `yRange` (Tuple[float, float]): Domain extent in y-direction (min_y, max_y)
- `AoA` (float): Angle of attack in degrees. Default: 0.0

8.3.3 Attributes

- `is2D`: Boolean flag for 2D simulation (always True)

- `problemTag`: Case identifier
- `mesh`: Instance of Mesh class for spatial discretization
- `model`: PINN model with architecture [20,40,60,40,20]
- `c`: Airfoil chord length (normalized to 1.0)
- `x0, y0`: Airfoil leading edge coordinates
- `xAirfoil, yAirfoil`: Arrays storing airfoil surface coordinates

8.4 Core Methods

8.4.1 generate_airfoil_coords

Generates coordinates for a NACA 4-digit airfoil profile.

```python
def generate_airfoil_coords(self, N=100, thickness=0.12)
```

Implementation Details

The method implements the NACA 4-digit series equation:

$$y_t = \frac{t}{0.2}\left(a_0\sqrt{x} - a_1 x - a_2 x^2 + a_3 x^3 - a_4 x^4\right) \tag{8.1}$$

where:

- t: Maximum thickness ratio (default: 0.12)
- Coefficients: $a_0 = 0.2969$, $a_1 = 0.1260$, $a_2 = 0.3516$, $a_3 = 0.2843$, $a_4 = 0.1015$

8.4.2 generateMesh

Creates the computational mesh including boundary points.

```python
def generateMesh(self, Nx: int = 100,
                 Ny: int = 100,
                 NBoundary: int = 100,
                 sampling_method: str = 'random')
```

Parameters

- `Nx, Ny`: Number of points in x and y directions
- `NBoundary`: Number of boundary points
- `sampling_method`: Point distribution method ('random' or 'uniform')

Boundary Conditions

The method implements the following boundary conditions:

- Inlet: Prescribed velocity $(u_\infty \cos(\alpha), u_\infty \sin(\alpha))$
- Outlet: Zero gradient conditions
- Top/Bottom: Free-stream conditions
- Airfoil Surface: No-slip condition $\mathbf{u} = 0$

8.4.3 _initialize_boundaries

Sets up boundary conditions for all domain boundaries.

```python
def _initialize_boundaries(self)
```

Boundary Types

- Exterior Boundaries:
  - Inlet: Dirichlet for velocity, Neumann for pressure
  - Outlet: Zero gradient conditions
  - Top/Bottom: Free-stream conditions
- Interior Boundaries:
  - Airfoil: No-slip wall with zero normal pressure gradient

8.4.4 train

Executes the PINN training process.

```python
def train(self, epochs=10000,
            print_interval=100,
            autosaveInterval=10000)
```

Training Process

- Initializes NavierStokesLoss object
- Minimizes combined loss including:
  - PDE residuals
  - Boundary condition satisfaction
  - Conservation of mass
- Automatically saves model at specified intervals

8.4.5 predict

Generates flow field solution using trained model.

```python
def predict(self) -> None
```

Output Fields

- u: x-velocity component
- v: y-velocity component
- p: pressure field

8.4.6 write_solution

Exports solution to CSV format.
def write$_solution(self, filename = None)$

8.5 Error Handling

The class implements comprehensive error checking:

- Input parameter validation
- Mesh generation verification
- Boundary condition consistency
- Solution field completeness
- File I/O operations

8.6 Usage Example

```python
# Initialize simulation
flow = FlowOverAirfoil(
    caseName="NACA0012",
    xRange=(-2, 4),
    yRange=(-2, 2),
    AoA=5.0
)

# Generate mesh
flow.generateMesh(Nx=200, Ny=200, NBoundary=150)

# Train PINN
flow.train(epochs=20000)

# Generate solution
flow.predict()

# Visualize results
flow.plot(solkey='u', streamlines=True)
```

8.7 Best Practices

1. Domain Size
    - Inlet boundary: 2-3 chord lengths upstream
    - Outlet boundary: 4-5 chord lengths downstream
    - Lateral boundaries: 2-3 chord lengths from airfoil
2. Training Parameters
    - Minimum recommended epochs: 10000
    - Regular model saving (every 1000-5000 epochs)
    - Monitor residuals for convergence

8.8 Performance Considerations

- Memory usage scales with $O(N_x \times N_y)$
- Training time depends on:
    - Mesh resolution
    - Neural network architecture
    - Number of training epochs
    - Hardware capabilities
- Solution accuracy affected by:
    - Boundary point distribution
    - Training convergence
    - Reynolds number regime

8.9 Validation

Recommended validation cases:

- NACA 0012 at $\alpha = 0°$
- Comparison with potential flow at low Reynolds numbers
- Boundary condition satisfaction check