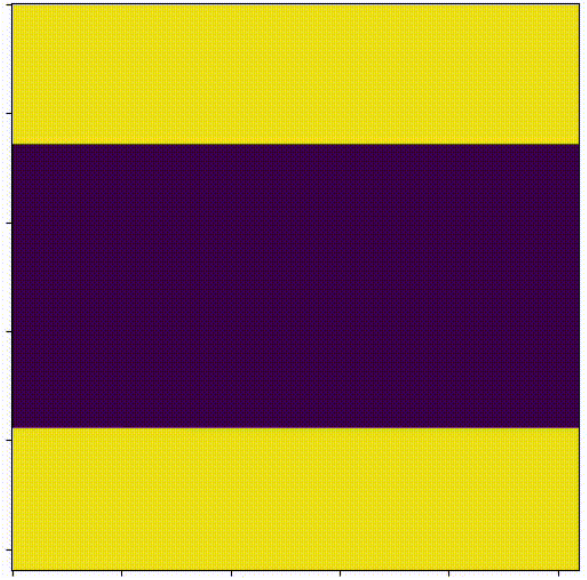
FIEFS: A Flow Instability Eulerian Flow Solver



APC524 Final Project

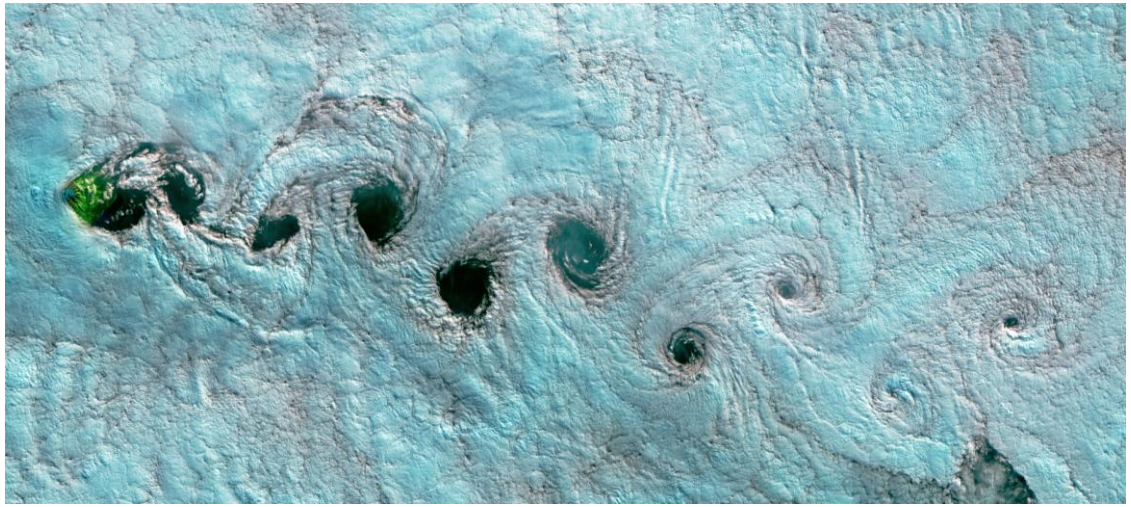
Michael D. Walker, Philip Satterthwaite
PhD Candidates
Mech. & Aero. Engineering
(mw6136, ps1639)

PRINCETON UNIVERSITY

Introduction

Computational fluid dynamics can reveal significant detail of the fundamental physical processes in a fluid flow as well as inform engineering design. In particular, flow instabilities important phenomena science and nature. The basic Euler solver presented in this report was adapted and enhanced from existing numerical schemes [1, 2] and applied to two test cases (Karman vortex generation from a bluff-body in channel flow, Kelvin-Helmholtz instability from velocity shear in channel flow).

Karman Vortex Street



Kelvin-Helmholtz Instability

[1] Boerchers et al. *PSYCHo-I*, (2022).

[2] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*, (2009).

Contributions

Michael:
- Adapted the **FIEFS** solver architecture, including improvements in speed and stability.
- Implemented the two test cases (Karman vortex generation and Kelvin-Helmholtz instability).
- Improved the data storage and visualization tools.
- Set up the initial framework for continuous integration.
- Relevant sections of the report and generated the figures shown. Created presentation slides.

Philip:
- Wrote and implemented the GUI into the **FIEFS** solver (including class structure and tests).
- Development of the testing and continuous integration framework.
- Wrote new tests and adapted old tests for new instabilities.
- Significant debugging of the solver and GUI.
- Relevant sections in the project report.

Euler Equations

Governing (conservation) equations for fluid flow:

Mass: $\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_j)}{\partial x_j} = 0$

Momentum (in 2-D): $\frac{\partial (\rho u_i)}{\partial t} + \frac{\partial (\rho u_j u_i)}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \rho g_i + \frac{\partial \tau_{ij}}{\partial x_j}$

$$\tau_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{2}{3}\mu\frac{\partial u_k}{\partial x_k}\delta_{ij}$$

Energy: $\frac{\partial (\rho h)}{\partial t} + \frac{\partial (\rho u_j h)}{\partial x_j} = \frac{\partial P}{\partial t} + \frac{\partial}{\partial x_j}\left(\lambda \frac{\partial T}{\partial x_j}\right) + \frac{\partial}{\partial x_j}\left(\rho \sum_{\alpha=1}^{N} D_\alpha \, h_\alpha \frac{\partial Y_\alpha}{\partial x_j}\right)$

The Euler equations neglect viscosity and body force terms, allowing for numerically feasible methods of solution

$$\frac{\partial}{\partial t}\begin{bmatrix} \rho \\ \rho u_i \\ \rho u_j \\ E \end{bmatrix} + \frac{\partial}{\partial x_i}\begin{bmatrix} \rho u_j \\ \rho u_i u_j \\ \rho u_j^2 + P \\ u_j(E+P) \end{bmatrix} + \frac{\partial}{\partial x_j}\begin{bmatrix} \rho u_i \\ \rho u_i^2 + P \\ \rho u_i u_j \\ u_i(E+P) \end{bmatrix} = 0$$

C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics*, Cambridge Aerospace Series (1998).

Numerical Scheme

Implemented a second-order finite-volume MUSCL-Hancock scheme (Monotonic Upstream-Centered Scheme for Conservation Laws) in four steps:

1. Data Reconstruction: Cell-centered values are extrapolated into slopes between grid cells so that the fluid is represented by a piecewise-linear plane in each of the cells
2. Evolution: Advance the boundary extrapolated values by a half timestep
3. Riemann Solution: HLLC (Harten-Lax-van Leer-Contact) Riemann solver was implemented to calculate the fluxes in the *i* and *j* direction at the cell faces
4. Conservative Update: Update and store values of conserved fluxes.

A Courant-Friedrichs-Lewy (CFL) condition and smoothing factor (SF) were implemented to achieve numerical stability by imposing a limit on the maximum time step that can be used and limiting the flux residuals to ensure convergence.

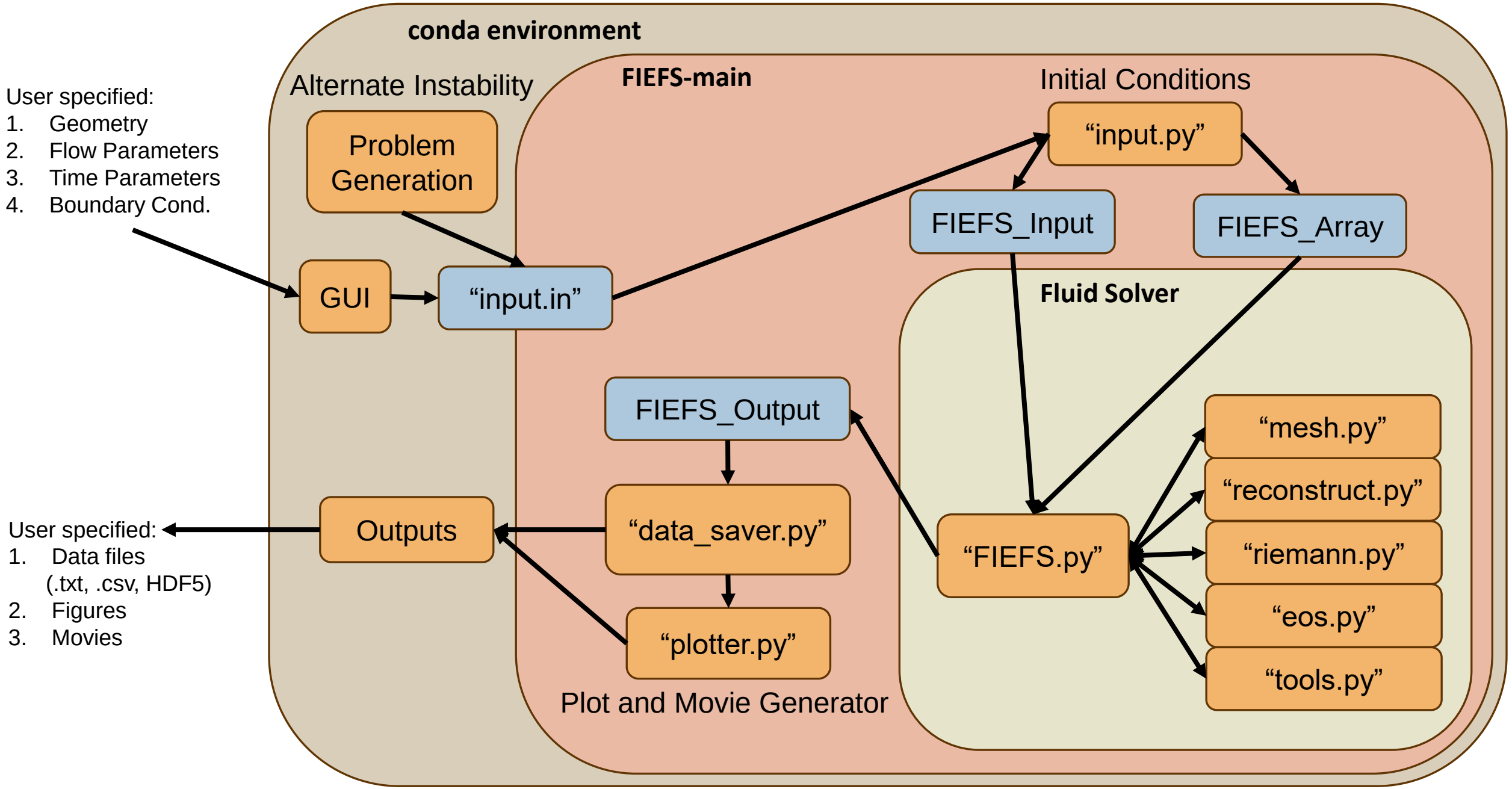
E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*, (2009).

Software Architecture

The basic program

- reads in geometry and flow data
- discretizes the geometric domain (grid generation)
- imposes the CFL number to set the time-step
- applies boundary conditions at the inflow and outflow of the domain, and wall
- sets and sums fluxes of the four Euler equations
- calculates secondary variables from the four conserved (density, velocity in 2-D, and internal energy)
- applies artificial smoothing

- The user defines the input files through a GUI. The **FIEFS_Input** class stores the parameter information in a dictionary that can be accessed by other elements of the code.
- The **ProblemGenerator** function that uses the input parameters in the **FIEFS_Input** object to initialize the simulation array (Un) in the **FIEFS_Array**. Once the **FIEFS_Array** is initialized, the initial conditions are set and the array can be passed into the fluid solver where the problem is subsequently evolved.
- The **FIEFS.py** solver and its helper functions are called to then execute a solution and update the values.


conda environment
Alternate Instability
FIEFS-main
Initial Conditions
User specified:
1. Geometry
2. Flow Parameters
3. Time Parameters
4. Boundary Cond.
Problem Generation
"input.py"
FIEFS_Input
FIEFS_Array
GUI
"input.in"
Fluid Solver
FIEFS_Output
"mesh.py"
"reconstruct.py"
"riemann.py"
"eos.py"
"tools.py"
Outputs
"data_saver.py"
"FIEFS.py"
"plotter.py"
Plot and Movie Generator
User specified:
1. Data files
(.txt, .csv, HDF5)
2. Figures
3. Movies

Primary Class Structures (input)

class input.**FIEFS_Input(input_fname: str)**

Bases: object

Class containing the input information

PARAMETERS:

input_fname (*str*) – The name of the input file

input_fname

The name of the input file

TYPE: str

value_dict

Dictionary containing the problem input information (after parsing the input file)

TYPE: dict

parse_input_file() → None

Parses and stores information from input file

Loops through parameter file and stores values from file to be used in problem generator

See end of presentation for full list of class structures

Generated with Sphinx

Primary Class Structures (solver)

class mesh.**FIEFS_Array(pin: FIEFS_Input, dtype: dtype)**

Bases: object

Class which contains the mesh and conserved variables

PARAMETERS:

- **pin** (*FIEFS_Input*) – Contains the problem information stored in the FIEFS_Input object
- **dtype** (*dtype*) – Specify the dtype for the conserved variables to be stored in the PyschoArray

nvar: Number of variables to be stored

TYPE: int

nx1: Number of cells in the x1 direction

TYPE: int

nx2: Number of cells in the x2 direction

TYPE: int

ng: Number of ghost cells

TYPE: int

x1min, x2min: Min x1 and x2 values

TYPE: float

x1max, x2max: Max x1 and x2 values

TYPE: float

dx1, dx2: Step size in the x1 and x2 directions

TYPE: float

Un: Conserved variables

TYPE: ndarray[dtype]

enforce_bcs(pin: FIEFS_Input) → None

Implements the desired boundary conditions

Will enforce the boundary conditions set in the FIEFS_Input class on the conserved variables, Un.

pinFIEFS_Input

Contains the problem information stored in the FIEFS_Input object

print_value(indvar: int, indx1: int, indx2: int) → None

Generated with Sphinx

Primary Class Structures (output)

class data_saver.**FIEFS_Output(input_fname: str)**

Bases: object

Class for producing the desired output

Produces the desired output variables in the desired format - set in the problem input

input_fname

File name for the output file

TYPE: str

data_preferences(pin: FIEFS_Input) → None

This function is called in *FIEFS.py* and sets the data preferences specified in the problem input (pin).

PARAMETERS:

pin (*FIEFS_Input*) – Contains the problem information stored in the FIEFS_Input object

save_data(pmesh: FIEFS_Array, t: float, tmax: float, gamma: float, iter: float) → None

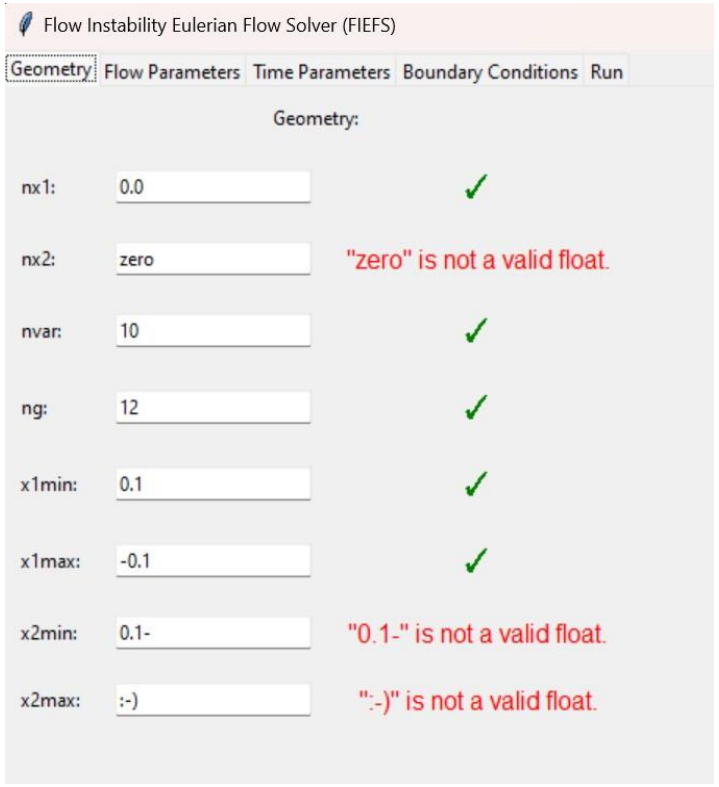
Saves the data to to the desired format

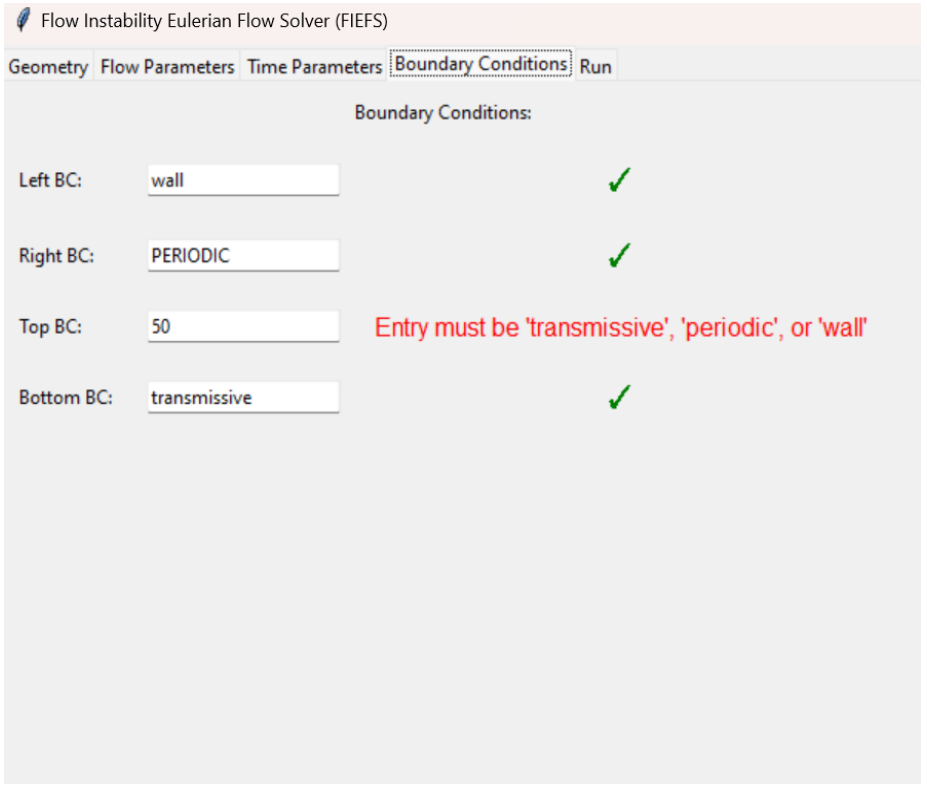
PARAMETERS:

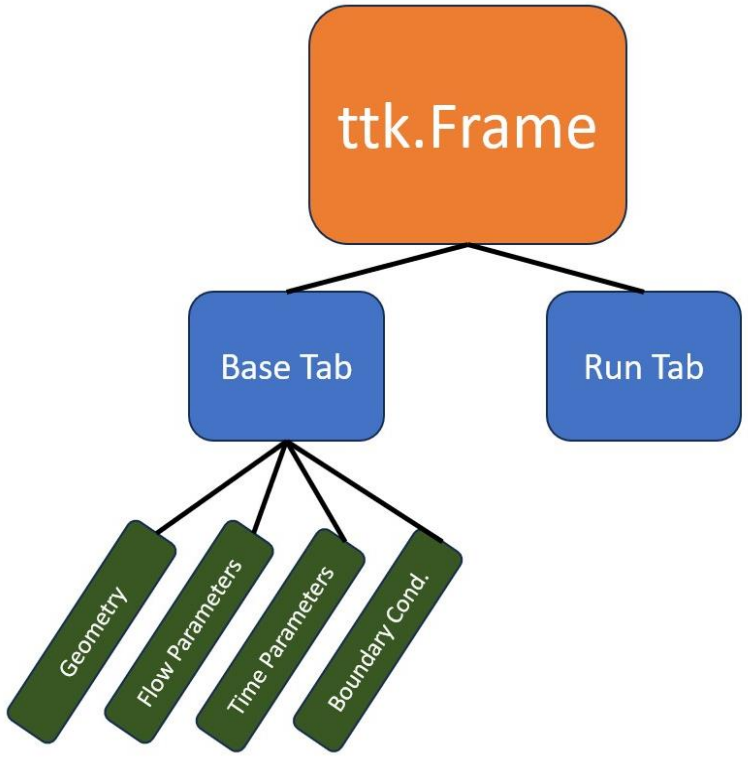
- **pmesh** (*FIEFS_Array*) – FIEFS_Array mesh which contains all of the mesh information and the conserved variables, Un
- **t** (*float*) – The current time
- **tmax** (*float*) – The maximum time or the time the simulations runs until
- **gamma** (*float*) – Specific heat ratio

Generated with Sphinx

Object-Oriented Graphical User Interface (GUI)





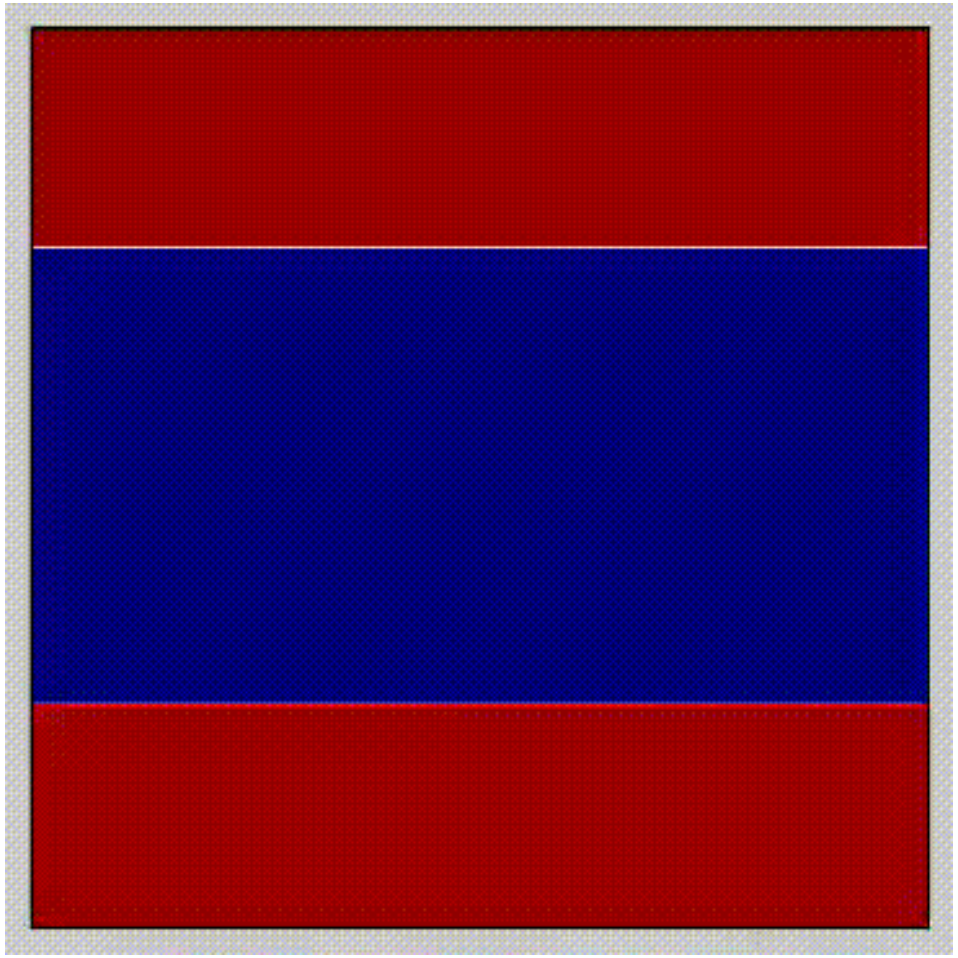


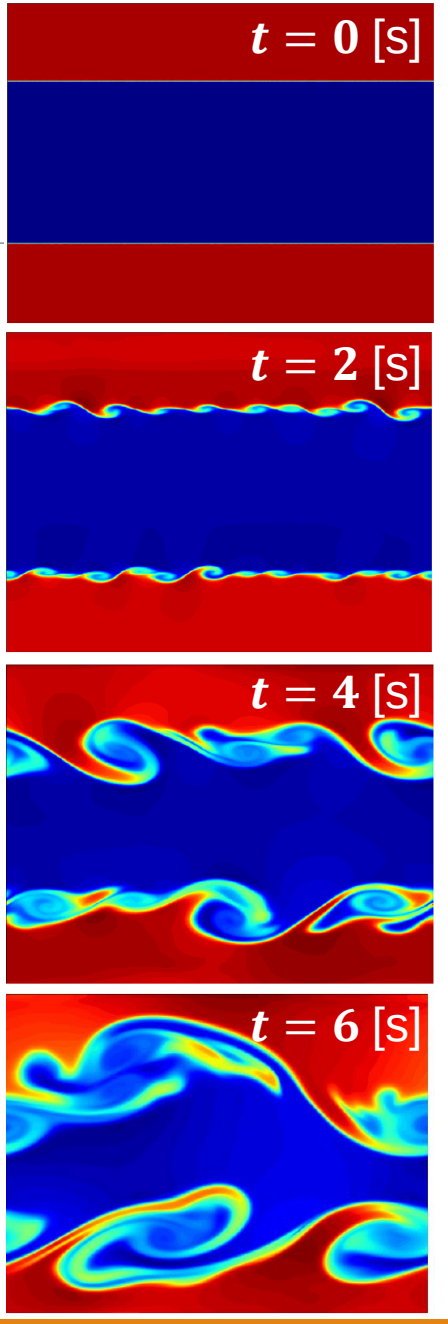
Results

Kelvin-Helmholtz Instability in Velocity Shear

Kelvin-Helmholtz Instability
(t = 0.000 sec)

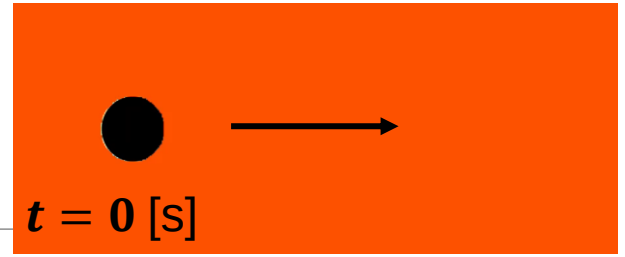
.gif

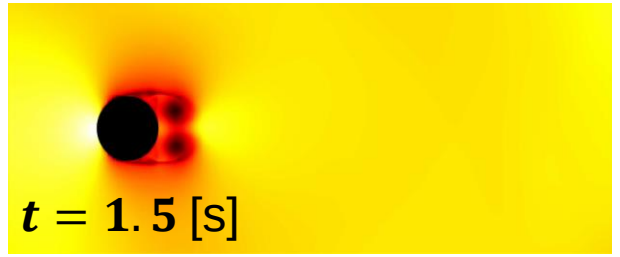


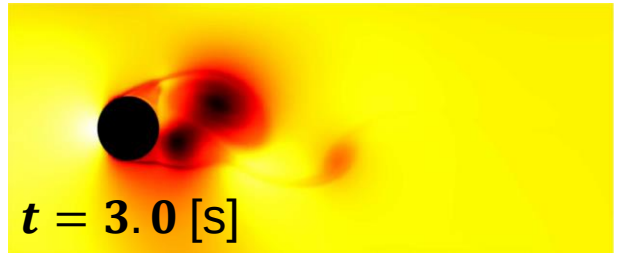


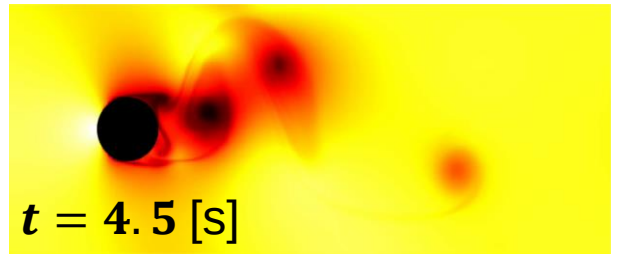
Kármán Vortex Generation from Bluff Body

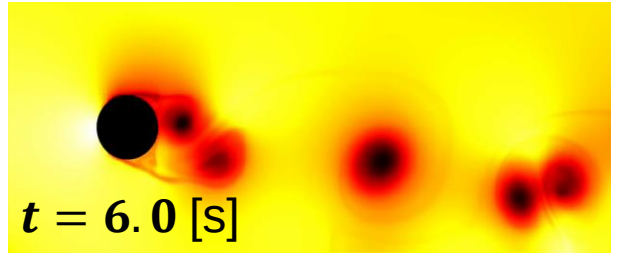
.gif











Version Control and Testing

- Version control implemented using **Git**, allowing for branching and work in parallel. **Github** repository allowed further collaboration with “pull requests” and commit comments. Testing could be conducted and displayed remotely with “Github actions.”

- Several tests were already included in the parent repository. These were reformatted for proper implementation using **nox** virtual environments and **Github** CI.
- A series of tests were developed to test the error-output boolean logic for verification of user input values.

Documentation and Continuous Integration

- **Conda** was used to create the environment with the necessary packages and dependencies with “environment.yaml”.

- **Pre-commit** was used to handle the linting and automatically resolve or notify for formatting, style, or bug-prone related issues in “.pre-commit-config.yaml” with standard hooks, calling automatic formatters and code linters such as, **Ruff**, **Flake8**, and **PyUpgrade.**
- “ci.yml” calls for pre-commit and testing processes to be run
- **Nox** generates the test environment and “noxfile.py” runs code-specific tests that are verified to succeed on **Python** versions 3.8, 3.9, 3.10, and 3.11

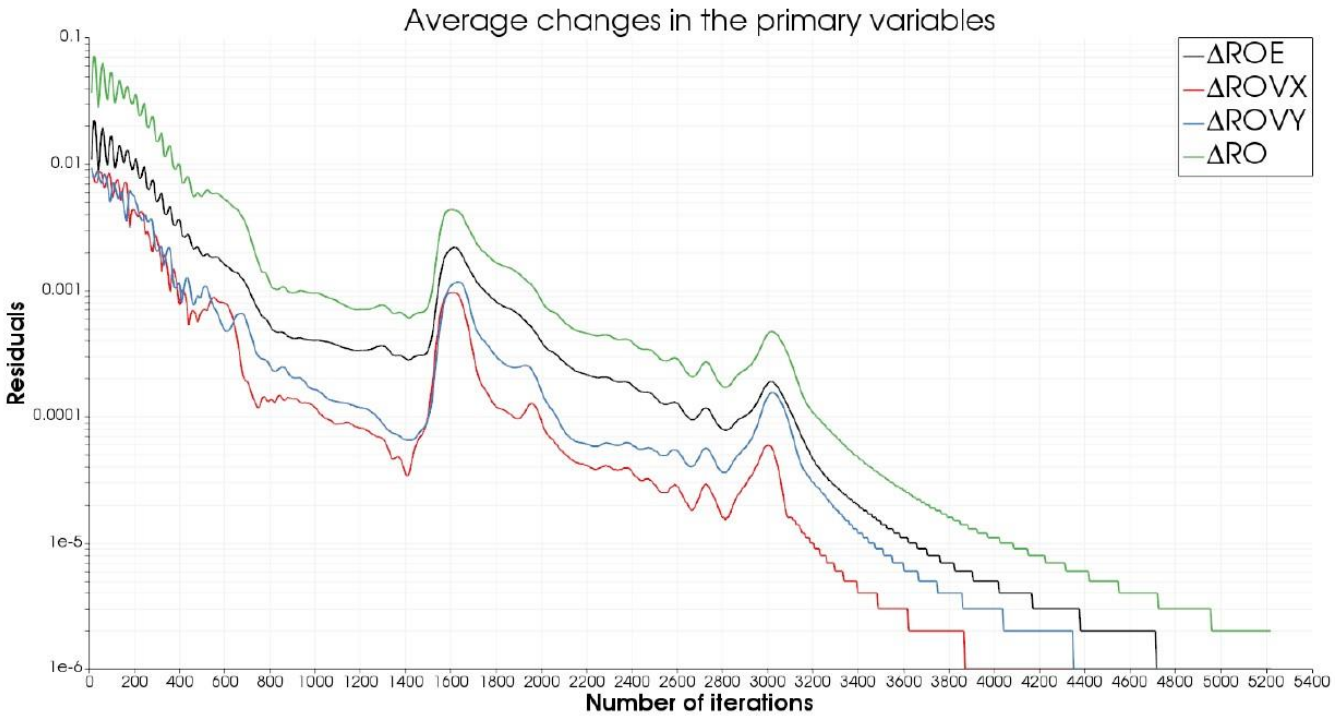
- Documentation of the functions and classes was generated using **Sphinx**

Performance Engineering

- For both implemented problems, the solver achieves adequate convergence in ~1000 timesteps. Mesh optimization was beyond the scope of this project.

- Performance was optimized by limiting the number of looping functions and using **NumPy** vectorization wherever possible.

- Whenever major looping was required, **Numba** just-in-time compiler was used in “no Python mode”.



Future Work

- Numerical Improvements: **Speed** (constant stagnation enthalpy, spatially varying timesteps, residual averaging), **Accuracy** (higher-order smoothing, higher-order differencing, differed correction factors), **Stability** (high-order Runge-Kutta and second-order term in time derivative)

- New Cases: Implement additional instabilities and boundary conditions (Rayleigh-Taylor, Richtmeyer-Meshkov, Rayleigh-Benard Convection)

- Testing: verification of mass conservation, time-based tests added to CI for performance engineering

- GUI: Add program status information and plotting tools

Lessons Learned

- Gained significant proficiency in virtually all topics of this course.
- Git is an essential tool for collaboration and version control, but real-world communication is essential.
- Collaborate on writing code in a “linear” way when possible, such that merge conflicts are straightforward to resolve.
- Ideas of testing and performance must be kept in mind from the beginning (test-driven development!)
- We will start implementing more tests and CI into the flow solvers our research group uses (the Computational Turbulent Reacting Flow Laboratory). We have real problems with version control and allowing bugs to propagate through different branches.

Additional Class Structures

eos.**e_EOS(rho: Union[float, ndarray], p: Union[float, ndarray], gamma: float) → Union[float, ndarray]**

Equation of state for internal energy

PARAMETERS:

- **rho** (*Union[float, ndarray]*) – Density
- **p** (*Union[float, ndarray]*) – Pressure
- **gamma** (*float*) – Specific heat ratio

RETURNS:

Internal Energy

RETURN TYPE:

Union[float, ndarray]

eos.**p_EOS(rho: Union[float, ndarray], e: Union[float, ndarray], gamma: float) → Union[float, ndarray]**

Equation of state for pressure

PARAMETERS:

- **rho** (*Union[float, ndarray]*) – Density
- **e** (*Union[float, ndarray]*) – Internal energy
- **gamma** (*float*) – Specific heat ratio

RETURNS:

Pressure

RETURN TYPE:

Union[float, ndarray]

tools.**calculate_timestep(pmesh: FIEFS_Array, cfl: float, gamma: float) → float**

Calculates the maximum timestep allowed for a given CFL to remain stable

PARAMETERS:

- **pmesh** (*FIEFS Array*) – FIEFS_Array mesh which contains all of the current mesh information and the conserved variables Un
- **cfl** (*float*) – Courant-Freidrichs-Lewy condition necessary for stability when choosing the timestep
- **gamma** (*float*) – Specific heat ratio

RETURNS:

The calculated timestep for the provided conditions

RETURN TYPE:

float

tools.**get_fluxes_1d(Un: ndarray, gamma: float, direction: str) → ndarray**

Returns fluxes provided the conserved variables, Un, at a point

This function returns the fluxes in the x or y direction at one specified cell provided the conserved variables, Un, at the specified cell

PARAMETERS:

- **Un** (*ndarray[float]*) – Conserved variables
- **gamma** (*float*) – Specific heat ratio
- **direction** (*str*) – Specify the 'x' or 'y' direction

RETURNS:

F – The computed flux vector at one cell in the specified direction

RETURN TYPE:

ndarray[float]

tools.**get_fluxes_2d(Un: ndarray, gamma: float, direction: str) → ndarray**

Returns fluxes provided the conserved variables, Un, for all cells

This function returns the fluxes in the x or y direction for all cells provided the conserved variables, Un.

PARAMETERS:

- **Un** (*ndarray[float]*) – Conserved variables
- **gamma** (*float*) – Specific heat ratio
- **direction** (*str*) – Specify the 'x' or 'y' direction

RETURNS:

F – The computed flux vector at all cells in the specified direction

RETURN TYPE:

ndarray[float]

mesh.**get_interm_array(nvar: int, nx1: int, nx2: int, dtype: dtype) → ndarray**

Generates empty scratch array for intermediate calculations

PARAMETERS:

- **nvar** (*int*) – Number of variables
- **nx1** (*int*) – Number of cells in the x1 direction
- **nx2** (*int*) – Number of cells in the x2 direction
- **dtype** (*dtype*) – Type for the values to have in the scratch array

RETURNS:

Scratch array with provided dtype

RETURN TYPE:

ndarray

reconstruct.**get_unlimited_slopes(U_i_j: ndarray, U_ip1_j: ndarray, U_im1_j: ndarray, U_i_jp1: ndarray, U_i_jm1: ndarray, w: float)**

Find the slopes between grid cells without a limiter. Can potentially lead to oscillations if there are discontinuities present in flow.

PARAMETERS:

- **U_i_j** (*ndarray[float]*) – Values of U not shifted by any index
- **U_ip1_j** (*ndarray[float]*) – Values of U shifted by i+1
- **U_im1_j** (*ndarray[float]*) – Values of U shifted by i-1
- **U_i_jp1** (*ndarray[float]*) – Values of U shifted by j+1
- **U_i_jm1** (*ndarray[float]*) – Values of U shifted by j-1
- **w** (*float*) – Weight parameter determining whether slopes are centered or biased in a particular direction

RETURNS:

- **delta_i** (*ndarray[float]*) – Slope of the conserved variables in the x-direction
- **delta_j** (*ndarray[float]*) – Slope of the conserved variables in the y-direction

reconstruct.**get_limited_slopes(U_i_j: ndarray, U_ip1_j: ndarray, U_im1_j: ndarray, U_i_jp1: ndarray, U_i_jm1: ndarray, beta: float)**

Minmod slope limiter to handle discontinuities

Minimod slope limiter based on page 508 in [1], necessary for handling discontinuities

PARAMETERS:

- **U_i_j** (*ndarray[float]*) – Values of U not shifted by any index
- **U_ip1_j** (*ndarray[float]*) – Values of U shifted by i+1
- **U_im1_j** (*ndarray[float]*) – Values of U shifted by i-1
- **U_i_jp1** (*ndarray[float]*) – Values of U shifted by j+1
- **U_i_jm1** (*ndarray[float]*) – Values of U shifted by j-1
- **beta** (*float*) – Weight value determining type of limiter. Default value is 1.0 which represents a minmod limiter

RETURNS:

- **delta_i** (*ndarray[float]*) – Slope of the conserved variables in the x-direction
- **delta_j** (*ndarray[float]*) – Slope of the conserved variables in the y-direction

tools.**get_primitive_variables_1d(Un: ndarray, gamma: float)**

Returns the primitive variables at a point provided Un.

This function returns the primitive variables at a single point provided the conserved variables Un are provided at the same point.

PARAMETERS:

- **Un** (*ndarray[float]*) – Conserved variables
- **gamma** (*float*) – Specific heat ratio

RETURNS:

- **rho** (*float*) – Density
- **u** (*float*) – Horizontal velocity
- **v** (*float*) – Vertical velocity
- **p** (*float*) – Pressure

tools.**get_primitive_variables_2d(Un: ndarray, gamma: float)**

Returns the primitive variables for all points provided Un.

This function returns the primitive variables for all points provided the conserved variables Un are provided.

PARAMETERS:

- **Un** (*ndarray[float]*) – Conserved variables
- **gamma** (*float*) – Specific heat ratio

RETURNS:

- **rho** (*ndarray[float]*) – Density
- **u** (*ndarray[float]*) – Horizontal velocity
- **v** (*ndarray[float]*) – Vertical velocity
- **p** (*ndarray[float]*) – Pressure

riemann.**solve_riemann(U_l: ndarray, U_r: ndarray, gamma: float, direction: str) → ndarray**

Solve the Riemann problem

Solves the Riemann problem using a HLLC Riemann solver - outlined in Toro adapted from page 322 (see [1])

PARAMETERS:

- **U_l** (*ndarray[float]*) – Conserved variables at the left cell face
- **U_r** (*ndarray[float]*) – Conserved variables at the right cell face
- **gamma** (*float*) – Specific heat ratio
- **direction** (*str*) – Specify the 'x' or 'y' direction

RETURNS:

F – The flux in the specified direction returned from the Riemann problem

RETURN TYPE:

ndarray[float]

kh.**ProblemGenerator(pin: FIEFS_Input, pmesh: FIEFS_Array) → None**

Generates the problem in by inputting the information to the problem mesh

This function is called in *main.py* and sets the initial conditions specified in the problem input (pin) onto the problem mesh (pmesh).

Needs to exist for each problem type in order for everything to work.

PARAMETERS:

- **pin** (*FIEFS_Input*) – Contains the problem information stored in the FIEFS_Input object
- **pmesh** (*FIEFS_Array*) – FIEFS_Array mesh which contains all of the current mesh information and the conserved variables Un

sample.**sampleProblemGenerator(pin: FIEFS_Input, pmesh: FIEFS_Array) → None**

Generates the problem in by inputting the information to the problem mesh
This function is called in *main.py* and sets the initial conditions specified in the problem input (pin) onto the problem mesh (pmesh).
Needs to exist for each problem type in order for everything to work.
PARAMETERS:

- **pin** (*FIEFS_Input*) – Contains the problem information stored in the FIEFS_Input object
- **pmesh** (*FIEFS_Array*) – FIEFS_Array mesh which contains all of the current mesh information and the conserved variables Un

class plotter.**Plotter(pmesh: FIEFS_Array)**

Bases: object

Creates plots for the desired variables

PARAMETERS:

pmesh (*FIEFS_Array*) – FIEFS_Array mesh which contains all of the current mesh information and the conserved variables Un

ng: Number of ghost cells

TYPE: int

rho: Array containing the rho (density) value at each point in space at a given time

TYPE: ndarray[float]

u: Array containing the u (horizontal velocity) value at each point in space at a given time

TYPE: ndarray[float]

v: Array continaing the v (vertical velocity) value at each point in space at a given time

TYPE: ndarray[float]

et: Array containing the et (total energy) value at each point in space at a given time

TYPE: ndarray[float]

primitives: Dictionary containing each primitive (rho, u, v, et) with a corresponding key

TYPE: dict

x1: Vector containing the x1 values

TYPE: ndarray[float]

x2: Vector containing the x2 values

TYPE: ndarray[float]

x1_plot: Array of x1 values after using meshrid for plotting

TYPE: ndarray[float]

x2_plot: Array of x2 values after using meshgrid for plotting

TYPE: ndarray[float]

check_path_exists() → None

Checks output path and creates it if necessary

create_plot(variables_to_plot: list[str], labels: list[str], cmaps: list[str], stability_name: str, style_mode: bool, iter: int, time: float) → None

Creates desired plots for provided input variables at a given time

PARAMETERS:

- **variables_to_plot** (*list[str]*) – A list of strings containing the variables to be plotted
- **labels** (*list[str]*) – The corresponding labels for the variables (can be in LaTeX form)
- **cmaps** (*list[str]*) – The desired cmaps for each variable - entered in the corresponding order
- **stability_name** (*str*) – The name of the instability
- **style_mode** (*bool*) – Style_mode will turn off all extra text and display only the desired variable in full in the figure
- **iter** (*int*) – The current iteration
- **time** (*float*) – The current time