Michael D. Walker, Philip Satterthwaite
APC524: Software Engineering for Scientific Computing
December 15, 2023

Final Project Report: FIEFS
A Flow Instability Eulerian Flow Solver

Introduction

Background Computational fluid dynamics (CFD) provides effective means for the rapid evaluation of design scenarios and can produce significant detail of the fundamental physical processes in a flow situation [5, 6]. As a result, there is great motivation to develop low cost tools that enhance education and design. Experimental and prototype setups are often expensive and test conditions are difficult to replicate; numerical solutions provide a promising and reliable alternative. We continue the development of a compressible flow solver of the Euler equations, the inviscid Navier-Stokes formulations, applied to 2-D domains of several test cases and implementing several numerical schemes to improved speed, accuracy, and stability.

Project Statement The basic Euler solver presented in this report was adapted and enhanced from two existing numerical schemes [7, 1]. The advanced solver is then applied to two test cases (Kármán vortex generation from a bluff-body in channel flow, Kelvin-Helmholtz instability from velocity shear in channel flow). In addition, computational cost and speed are assessed based on the number of steps to convergence and clock runtime. The accuracy was assessed using the inlet-to-outlet mass flow ratio, the entropy generation, analytical comparison, and comparison to accepted CFD solutions. Other focuses of this project included improvement of user experience (and thus reduction of possible user errors) as well as improved code formatting, testing, and continuous integration.

The Software repository submitted with this report (https://github.com/mw6136/FIEFS) is the advanced solver with several test cases. Additional modifications are needed to specify new test cases, which are discussed. Figures were generated with `Python` and `Paraview`.

Michael's Contributions Michael adapted the `FIEFS` solver architecture, including improvements in speed and stability, as will be discussed. He implemented the two test cases (Kármán vortex generation and Kelvin-Helmholtz instability). He also developed the data storage and visualization tools and set up the initial framework for continuous integration. He contributed the relevant sections of the report and generated the figures shown. The majority of the presentation slides were also created by Michael.

Philip's Contributions Philip wrote and implemented the GUI into the `FIEFS` solver. This included developing a class structure to create the GUI as well as writing a series of tests to ensure that the GUI input verification scripts worked correctly. Philip also spent a significant amount of time debugging and implementing the testing and continuous integration framework. He also wrote relevant sections in the project report.

Physical Basis and Numerical Methods

Governing Equations in Fluid Mechanics Fluid flows present a non-linear, multi-scale problem described by the inherently coupled system of Navier-Stokes equations. These differential equations define conservation of mass and momentum assuming the continuum hypothesis [2]. The conservation of mass principle (continuity equation) is written as

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

where ρ is the density of the fluid and u_j is the velocity in the principal direction j.

Understanding the variation of velocities and scalars (e.g., ρ or T) in the flow at all conditions is necessary to describe the flow structure. The conservation of momentum is described by the Navier-Stokes transport equation

$$\frac{\partial(\rho u_i)}{\partial t} + \frac{\partial(\rho u_j u_i)}{\partial x_j} = -\frac{\partial P}{\partial x_i} + \rho g_i + \frac{\partial \tau_{ij}}{\partial x_j} \ ,$$

where u_i is the velocity in the principal direction i, P is the pressure at each point in the fluid, g_i is the body force in direction i, and τ_{ij} is the viscous shear stress. For constant-property Newtonian fluids, τ_{ij} is related to strain rate by

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

where u_k is the velocity in the principal direction k, μ is the dynamic viscosity of the fluid, and δ_{ij} is the Kronecker delta function.

The conservation of energy equation exists in many forms, but is presented here in terms of enthalpy for an arbitrary number of chemical species [2] as

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

where T is the temperature of the fluid, λ is the thermal conductivity, Y_α is the mass fraction of species α, h_α is the absolute enthalpy of species α, N is the total number of species in the mixture, D_α is the diffusion coefficient for species α, where Fick's law for diffusion is assumed, and h is the mixture enthalpy $\sum_{\alpha=1}^{N} Y_\alpha \ h_\alpha$. This formulation neglects radiation, and viscous heating and acoustic interactions under a low-speed assumption [2]. For a homogeneous non-chemically reacting flow, it can be assumed $\alpha = 1$ and $Y_\alpha = 1$.

An accompanying appropriate equation of state is required to close this system of equations. This usually takes the form of the ideal gas law, $P = \rho RT/\overline{W}$, where R is the universal gas constant and $\overline{W}$ is the mean molecular mass of the mixture. This system is readily solved for laminar flows, but analytical solutions do not currently exist for unstable or turbulent flows. There are multiple strategies to model such flows by solving a form of these equations numerically.

The Euler Equations A solution, either analytical or numerical, to the coupled Navier-Stokes system, remains elusive [4]. There is a well-known closure problem (more unknowns than equations). In certain circumstances, an appropriate simplification is to neglect the viscous and body force terms (i.e., $g_i = 0$ and $\mu = 0$), and apply a separate model of artificial smoothing. In a two-dimensional coordinate system, an inviscid, non-thermally conducting compressible flow can be modeled in this way by the Euler equations [8]. This system of equations is shown below in matrix form (for principle directions i and j), where E is the total energy.

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u_i\\ \rho u_j\\ E\end{bmatrix} + \frac{\partial}{\partial x_i}\begin{bmatrix}\rho u_j\\ \rho u_i u_j\\ \rho u_j^2 + P\\ u_j(E+P)\end{bmatrix} + \frac{\partial}{\partial x_j}\begin{bmatrix}\rho u_i\\ \rho u_i^2 + P\\ \rho u_i u_j\\ u_i(E+P)\end{bmatrix} = 0 \tag{1}$$

For brevity, $U \equiv [\,\rho\,, \rho u_i\,, \rho u_i\,, E\,]$. The Euler equations neglect fluid viscosity and thus do not require boundary layer modeling. However, they require other conditions to enforce stability. Effects of turbulence and Reynolds decomposition are not considered.

While the inlet-to-outlet mass flow ratio is one way to assess the quality of the solution, it is not always reliable. The flow field can be incorrect despite a ratio near unity. Entropy generation in the system can be computed as a better measure of accuracy. The total entropy change across the system, that is, between the inlet and outlet, can be estimated according to Equation 2 using the mean static temperature and

pressure values from the inlet and summing the entropy across each j line [8].

$$\Delta s = \sum_{j_{outlet}} c_p \cdot \ln\left(\frac{T_{j,outlet}}{\overline{T}_{static,inlet}}\right) - R \cdot \ln\left(\frac{P_{j,outlet}}{\overline{P}_{static,inlet}}\right) \tag{2}$$

Entropy generation was also evaluated between each grid point in Equation 3. This is under the assumption that entropy is being advected along the j lines, like streamlines.

$$\Delta s_{i,j} = c_p \cdot \ln\left(\frac{T_{i,j}}{T_{i-1,j}}\right) - R \cdot \ln\left(\frac{P_{i,j}}{P_{i-1,j}}\right) \tag{3}$$

MUSCL-Hancock scheme To numerically solve the Euler equations, a second-order finite-volume MUSCL-Hancock scheme (Monotonic Upstream-Centered Scheme for Conservation Laws) was first implemented by Broechers et al [1] from Toro [7]. The steps of this solver include:

1) Data Reconstruction
2) Evolution
3) Riemann Solution
4) Conservative Update

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 2-D cells. The equations describing cell slopes in the i and j direction are:

$$\Delta_i = \begin{cases} \mathbf{max}[0, \mathbf{min}(\beta\Delta_{i-1/2}, \Delta_{i+1/2}), \mathbf{min}(\Delta_{i-1/2}, \beta\Delta_{i+1/2})], & \Delta_{i+1/2} > 0 \\ \mathbf{min}[0, \mathbf{max}(\beta\Delta_{i-1/2}, \Delta_{i+1/2}), \mathbf{max}(\Delta_{i-1/2}, \beta\Delta_{i+1/2})], & \Delta_{i+1/2} < 0 \end{cases}$$

$$\Delta_j = \begin{cases} \mathbf{max}[0, \mathbf{min}(\beta\Delta_{j-1/2}, \Delta_{j+1/2}), \mathbf{min}(\Delta_{j-1/2}, \beta\Delta_{j+1/2})], & \Delta_{j+1/2} > 0 \\ \mathbf{min}[0, \mathbf{max}(\beta\Delta_{j-1/2}, \Delta_{j+1/2}), \mathbf{max}(\Delta_{j-1/2}, \beta\Delta_{j+1/2})], & \Delta_{j+1/2} < 0 \end{cases}$$

where $\Delta_{i\pm1/2}$, $\Delta_{j\pm1/2}$ are the slopes in i and j at the cell faces. `NumPy` vectorization was used to calculate these slopes, using a value of $\beta = 1$. The *boundary extrapolated values* (U_i^L, U_i^R, U_j^L, U_j^R) are then calculated.

$$U_i^L = U_{i,j}^n - \frac{1}{2}\Delta_i, \; U_i^R = U_{i,j}^n + \frac{1}{2}\Delta_i$$

$$U_j^L = U_{i,j}^n - \frac{1}{2}\Delta_j, \; U_j^R = U_{i,j}^n + \frac{1}{2}\Delta_j$$

Evolution: Each of the four boundary extrapolated values move forward by a half timestep.

$$\bar{U}_{i,j}^{L,R} = U_{i,j}^{L,R} + \frac{1}{2}\frac{\Delta t}{\Delta x}\left(\begin{bmatrix} \rho u_j \\ \rho u_i u_j \\ \rho u_j^2 + P \\ u_j(E+P) \end{bmatrix}_i^L - \begin{bmatrix} \rho u_j \\ \rho u_i u_j \\ \rho u_j^2 + P \\ u_j(E+P) \end{bmatrix}_i^R + \begin{bmatrix} \rho u_i \\ \rho u_i^2 + P \\ \rho u_i u_j \\ u_i(E+P) \end{bmatrix}_j^L - \begin{bmatrix} \rho u_i \\ \rho u_i^2 + P \\ \rho u_i u_j \\ u_i(E+P) \end{bmatrix}_j^R\right)$$

The Riemann problem can now be solved using the following equations for the left and right states:

$$U_{i,riemann}^L = \bar{U}_i^R, \; U_{i,riemann}^R = \bar{U}_{i+1}^L$$

$$U_{j,riemann}^L = \bar{U}_j^R, \; U_{j,riemann}^R = \bar{U}_{j+1}^L$$

Riemann Initial Value Problem: A HLLC (Harten-Lax-van Leer-Contact) Riemann solver was implemented to calculate the fluxes in the i and j direction at the cell faces [7]. It determines the states of nearby cells and then calculates the variable fluxes across those cells using conservation laws.

Conservative Update: The conserved variables are then updated through the equation below. This is the final step of the MUSCL-Hancock scheme, and is repeated for each consecutive time step.

$$(4)\quad U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{\Delta t}{\Delta x}\left(\begin{bmatrix}\rho u_j\\ \rho u_i u_j\\ \rho u_j^2 + P\\ u_j(E+P)\end{bmatrix}_{i-\frac{1}{2}} - \begin{bmatrix}\rho u_j\\ \rho u_i u_j\\ \rho u_j^2 + P\\ u_j(E+P)\end{bmatrix}_{i+\frac{1}{2}}\right) + \frac{\Delta t}{\Delta y}\left(\begin{bmatrix}\rho u_i\\ \rho u_i^2 + P\\ \rho u_i u_j\\ u_i(E+P)\end{bmatrix}_{j-\frac{1}{2}} - \begin{bmatrix}\rho u_i\\ \rho u_i^2 + P\\ \rho u_i u_j\\ u_i(E+P)\end{bmatrix}_{i+\frac{1}{2}}\right)$$

CFL Condition and Smoothing The Euler solver incorporates a Courant-Friedrichs-Lewy (CFL) condition and smoothing factor (SF) to achieve numerical stability at the expense of accuracy [8]. The CFL condition is a stability condition for numerical schemes that model wave phenomena. For an explicit time-marching simulation, the CFL condition imposes a limit on the maximum time step that can be used for a stable numerical simulation based on the minimum length scale in the numerical domain and the maximum convection speed, including flow velocity and acoustic wave propagation [3]. For non-reactive flows, the time step is not limited by the chemical reaction rates, so the acoustic wave propagation drives the CFL condition.

Viscosity is a natural form of damping in a fluid flow, smoothing the velocity and scalar fluctuations. While neglecting viscosity allows solving the conservation equations to be tractable, the fluxes will not converge. The smoothing factor is a second-order smoothing, which can be interpreted as adding artificial viscosity to the solution which degrades the accuracy of the result. However, this smoothing is necessary to maintain stability, as the smoothing factor adds to the value at some contribution from its surrounding cells. Thus, the higher the smoothing factor, the more stable but less accurate the solution.

Software Architecture

Code Framework We adhered to customary programming style and hygiene, as practiced in this course. The framework was designed to be robust, modular, and simple. This included easy implementation of new problems and independent input/output (i.e., pre-processing and post-processing outside of the solver).

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 ones (density, velocity in 2-D, and internal energy), and applies artificial smoothing. This was primarily accomplished through the `FIEFS_Input` and `FIEFS_Array` classes. Once these are fully defined, the solver has everything can resolve the results of the instability. The `FIEFS.py` solver and its helper functions are called to execute a solution and update the values. Figure 1 is a schematic representation of the `FIEFS` code.

The user defines the input files through a GUI (to be discussed in the next section), which contain the parameters and initial values that define the geometry, initial conditions, and boundary conditions. `Conda` was used to create the environment with the necessary packages and dependencies with "environment.yaml". The solver outputs are also user-specified, and include options for variable selection, output frequency, and file format. To demonstrate the project, a Kelvin-Helmholtz (K-H) instability and a Kármán vortex generation problem were implemented. The Kelvin-Helmholtz instability was simply a re-verification of previous developers to ensure that our changes to the file organization did not impact code functionality. Kármán vortex generation results from a bluff-body in channel flow due to aerodynamic flow and pressure changes caused by the resulting eddy. Implementing this new case was a verification of the capability of the solver to resolve new flow cases.

Graphical User Interface Development The graphical user interface (GUI) implemented in this code aims to create a streamlined user experience when creating the `FIEFS` input file. The user can quickly and easily alter all of the needed input parameters in one place by clicking through the tabs and changing parameters as needed. The default values are those in the Kelvin-Helmholtz instability. Inputs include

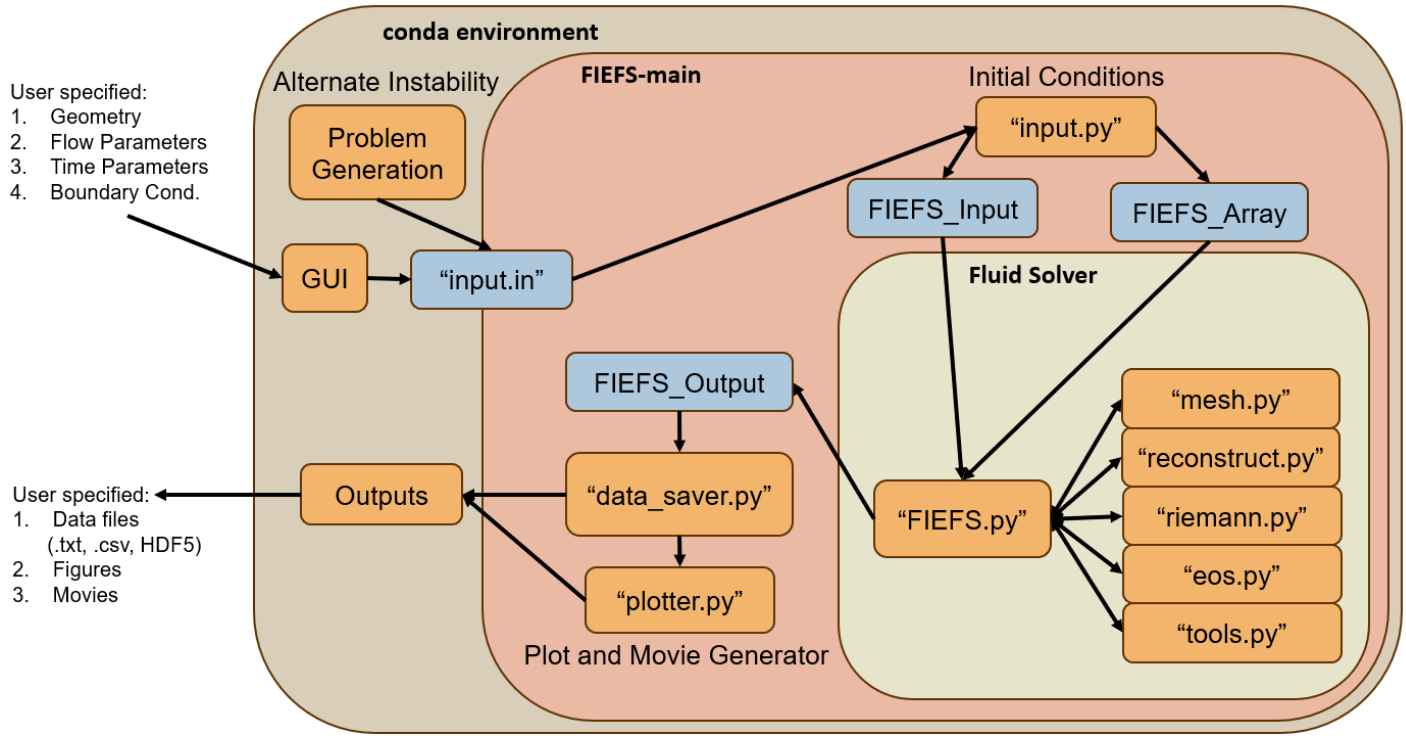

FIGURE 1. Diagram of the primary software functions and class structures. The problem is specified by the user through the GUI. This initializes input variables and passes them to the solver in the virtual environment. Outputs are post-processed as specified by the user.

specifications of the grid geometry, flow parameters, time parameters, and boundary conditions. The final tab of the GUI allows the user to run the flow solver and view the outputs. Thus, where the user previously had to open three files (input file, command terminal, and output file), the entire process can be completed from one place. The realization of this feature is unfortunately still one step away, as running terminal commands from a `Python` script varies significantly between operating systems. Therefore, the GUI currently only outputs the Linux terminal commands that the user should enter.

Perhaps the most useful aspect of the GUI is that it significantly limits the potential for user error when creating an input file in two ways. First, the `input.in` file is updated automatically. The correct values are inserted in the correct locations, eliminating potential user errors such as accidentally deleting or duplicating lines. Second, the GUI checks for user input errors. For example, it checks that all of the entered geometry, flow, and time parameters are all floats in the correct numerical range, and that the boundary conditions are one of the available options. The input file is updated only if the user enters a valid value, and is otherwise left as the default. Examples of these visuals can be seen in Figure 2.

A class structure is implemented in the GUI for ease of implementation of new features in the code. All classes inherit from `Python`'s "ttk.Frame" class. The `RunTab` class implements additional functions for generating the tab as well as for the execution of commands when the RUN button is selected. The `BaseTab` class has a slightly more complex structure. First, it requires more initialization inputs, namely title of the tab and a list of the required inputs as well as their corresponding location in the `input.in` file. Along with the initialization function (which also includes a call to a function that creates the widget), the class also implements functions that handle value entry. Upon a "value change" event, the `check_type` function is called to determine if the entered value is valid. If the value is invalid, the appropriate error message is displayed. If the value is valid, a green checkmark is displayed and the value is written to the correct location in the input file. The `GeometryTab`, `FlowParametersTab`, `TimeParametersTab`, and `BoundaryConditionsTab` are all subclasses of the `BaseTab` class. These subclasses each have different input parameters and different `check_type` functions due to the different natures of the inputs (e.g., boundary conditions must be strings whereas density must be a positive real number).

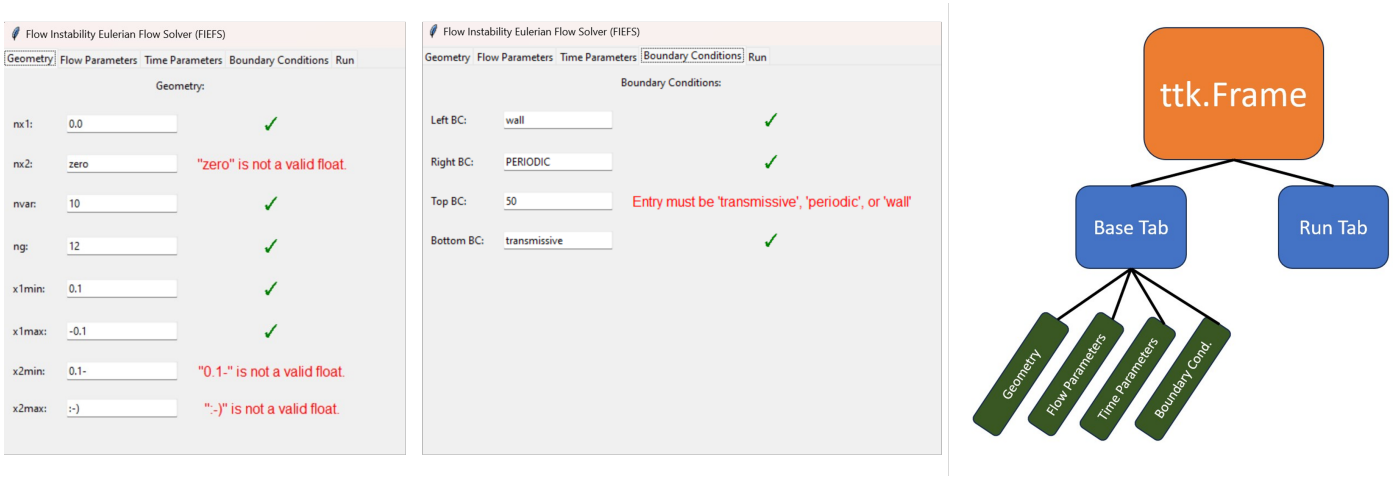

FIGURE 2. (left) GUI Grid Input Errors. (middle) GUI Boundary Condition Input Errors. (right) GUI Class Structure.

Data Storage and Visualization The `FIEFS_Output` class allows for the storage of data in several file formats. The method `data-preferences` reads the input file and stores the string inputs of variable and file type names as lists. The method `save-data` is called at the user-specified output frequency to store the iteration data in a format of the specified file type. The method `get-primitive-variables-2d` retrieves these stored arrays up to the maximum allowed time.

`FIEFS` culminates in the ability to visualize the computed simulation–a core feature of any CFD program–which allows for presentation and consideration of the physical phenomena. The plots are generated using the `Matplotlib` package, and the `plotter` class creates a subplot for each variable specified by the user, with data taken from `FIEFS_Array`. The test cases of Kelvin-Helmholtz instability and Kármán vortex generation were run to verify the functionality of storage and visualization, and are shown in Figure 3. Key errors were identified here; there is no way to specify the name or location of data storage, and the previous input and output files are rewritten each time a test case is run. These issues were aimed to be fixed through the GUI development.

Version Control and Testing In order to effectively collaborate on the project, version control was implemented using `Git`. This allowed each team member to work separately in parallel if necessary. Alternatively, it serves as a mechanism for providing modularity to the process through branching. The `Github` repository allowed further collaboration with "pull requests" and commit comments. Testing could be conducted and displayed remotely with "Github actions." The existing code framework was initially pushed to the `main` branch, with improvements to the solver made directly on `main`. The GUI framework was then developed on the `GUI` branch and the subsequent testing for it was created on `GUI_tests`. Finally, testing and continuous integration software debugging was performed on the `tests` branch. Each branch was then merged back into `main` after its development was finished.

The testing structure in `FIEFS` ensures the validity of the flow solver. Unit tests identify indexing errors by checking that newly defined arrays in functions were the dimension that they were intended to be. Further, each time a scalar value is recalculated, it is checked to be a positive value. Similarly unit tests for for the mesh ensure that there is no inconsistency between dimensions of arrays that to span the domain and those that to span the domain and ghost cells. Lastly the finite-difference approach interpolates values linearly between cell faces, thus the largest magnitude of these gradients for squares is one. The formatting and file structure were further modified to be compatible with `Github`'s continuous integration framework. The testing file was therefore restructured to function on a `nox` virtual environment.

Additionally, a series of tests were included in the `test_.py` file to ensure that the GUI logic performs correctly. As previously mentioned, the GUI automatically checks the validity of user input. The included tests verify that the appropriate functions within the GUI class structure return True or False based on

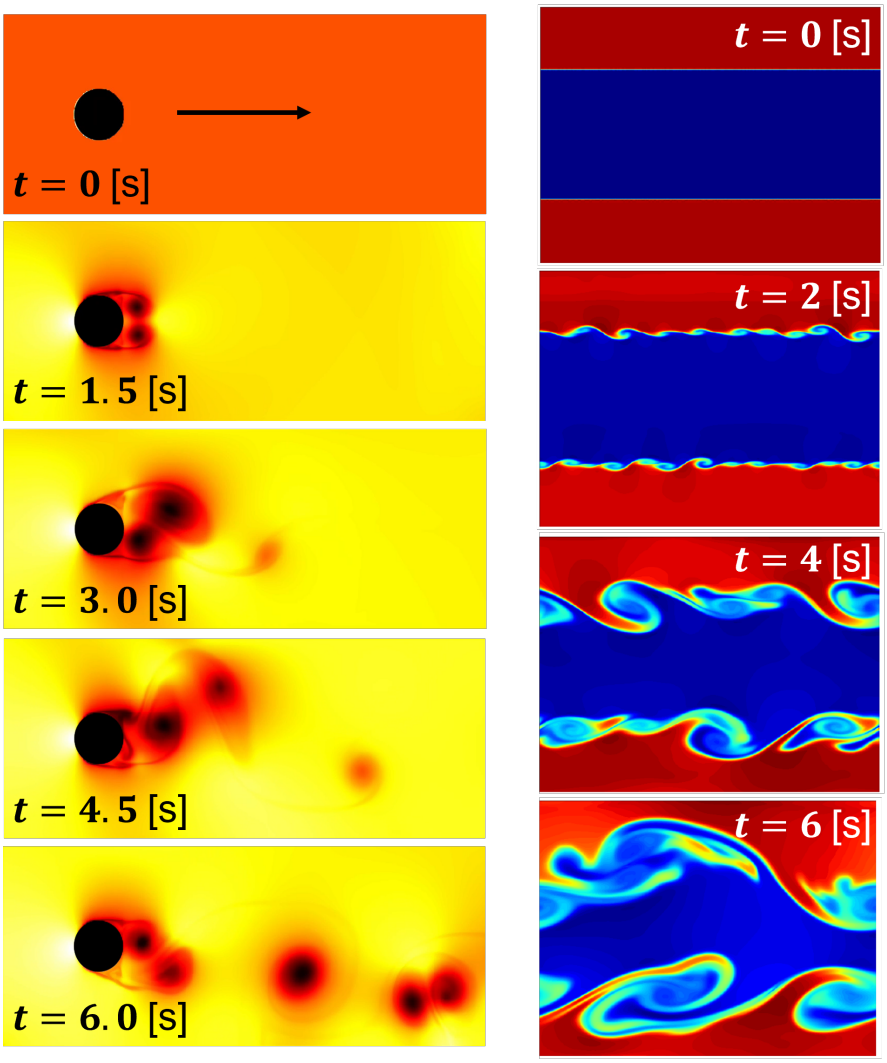

FIGURE 3. (left) Kármán vortex generation from bluff-body in channel flow. (right) Kelvin-Helmholtz instability from velocity shear in channel flow.

select possible user inputs. For example, it verifies that an entry "Wall" will return True for a boundary condition input, but returns False for a density or grid parameter input. In addition to type-checking, other tests were implemented to ensure that certain values cannot be negative, that the input is case insensitive. Due to the difficulty of creating GUI classes in a virtual environment, a creative restructuring of the `FIEFS` GUI class structure was needed such that certain functions only consider input logic where others modify GUI elements.

Continuous Integration Testing and continuous integration files were implemented to proactively address debugging and to ensure consistent file formatting. Continuous integration is implemented through "Github actions," and the processes are specified in the `ci.yml` file. The file includes calls to run two actions, namely the pre-commit functions and the testing file, via `nox`. These CI processes are specified to run whenever a push is made to any of the four branches of the repository. Results of the CI process as well as the entire history of debugging the CI implementation can be found in the "Actions" tab of the `GitHub` repository.

The ".pre-commit-config.yaml" file specifies what is checked in the pre-commit process. This includes standard pre-commit hooks such as trailing-whitespace, check-added-large-files, debug-statements, etc. The file also includes calls for debuggers, file formatters, and linters such as `ruff`, `black`, and `flake8`, as well as `PyUpgrade` to automatically update outdated `Python` syntax.

The most important function of the `noxfile.py` is that it includes a call to run tests created for this project. A virtual environment is created on `Python` versions 3.8, 3.9, 3.10, and 3.11, and each test is then run on each version, verifying that anyone who attempts to access and run the pushed code on their local environment will not come across any errors. Documentation of the functions and classes was generated using `Sphinx` and included as a .PDF file in the repository.

Performance Engineering The computational cost of this numerical simulation is determined largely by the resolution requirements—the solution domain must be large enough to contain the energy-containing motions, and the grid spacing must be small enough to resolve the dissipative scales. Methods for more efficient mesh discretization were beyond the scope of this project. Solution convergence for the K-H case is shown in Figure 4. Performance was optimized by limiting the number of looping functions and using vectorization wherever possible. The `FIEFS_Input` and `FIEFS_Array` classes, once initialized and filled, contained all information regarding parameters of the simulation, as well as a `NumPy` array containing the conserved variables of the fluid. The `FIEFS.py` solver and its helper functions are called to then execute a solution and update the values. Only in the case of implementing the HLLC Riemann solver in `FIEFS-main`, required an explicit double for loop over the computational grid, as `NumPy` vectorization was too difficult. Here the `Numba` just-in-time compiler was used in "no Python mode". With the addition of this feature, the code was significantly accelerated and much larger calculations could be achieved.

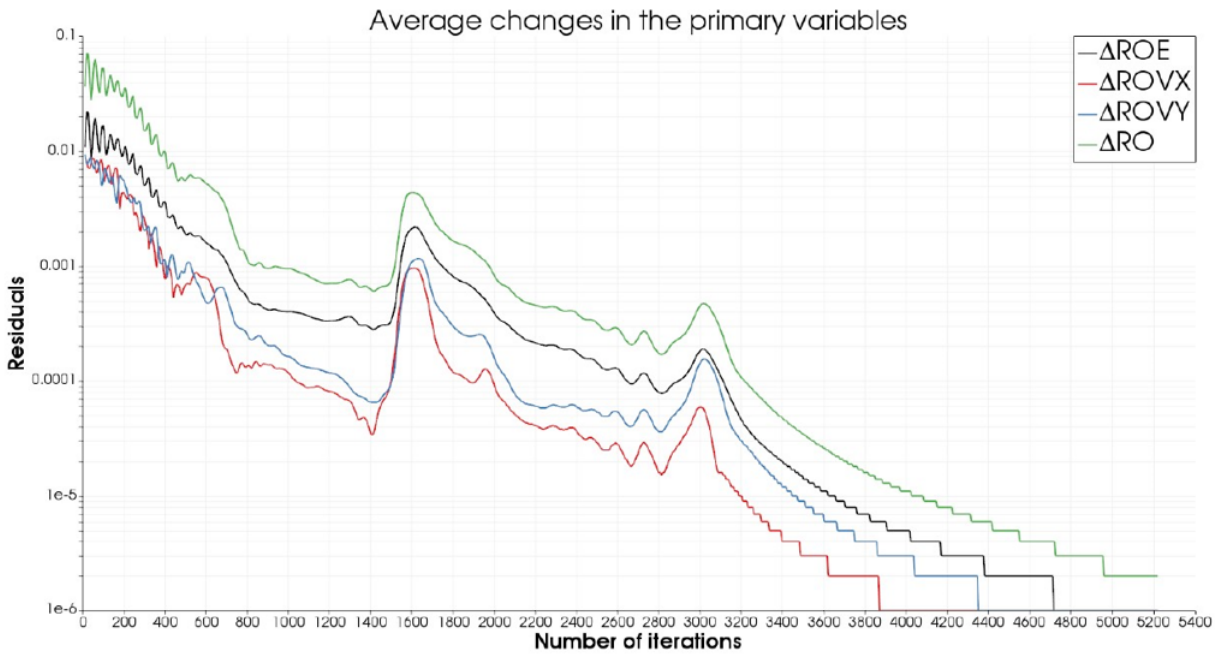

FIGURE 4. Flux Residuals for Kelvin-Helmholtz instability. Adequate convergence is shown by 1000 timesteps.

Future Work The are significant opportunities for improvements to the Euler solver in speed, stability, and accuracy. Speed could be achieved with the assumption of constant stagnation enthalpy for an adiabatic flow, spatially varying timesteps, or residual averaging. These would all likely degrade accuracy to some degree. Stability can be improved by using high-order Runge-Kutta and inclusion of a second-order term in the time derivative. Accuracy could be improved with techniques that limit the need for artificial smoothing, like using higher-order smoothing, higher-order differencing, or differed corrections.

Additional tests should also be developed that extend beyond the tests specific to the Kelvin-Helmholtz instability. This could include tests on integration processes, convergence criteria, and general mathematic boundary condition requirements needed to fully define a system. Time-based tests could also be added to see how code changes affect speed, to aid in performance engineering.

In terms of the `FIEFS` GUI, there are significant future developments that could further expedite the analysis process of the solver. First, some sort of progress bar or active display of solver residuals would enhance user experience. Second, a tab where the user can actively view and modify the plotted outputs of the simulation would be highly beneficial; the current method of viewing plots is to search through a file of hundreds of images, and data can only be re-processed by changing the input file and running the entire solver. Finally, the current program simply rewrites the output files each time an input is run. It would be wise to implement an option in the GUI where the user could save their input file with a specific name and also select the output name and location of the solver output data. The class structure of the GUI program would make the implementation of these features relatively simple.

Conclusion and Lessons Learned This project allowed us to develop further proficiency in virtually all aspects of the software engineering topics of this course. We learned that while `Git` is an essential tool for collaboration and version control, real-world communication is essential. Further, one must be deliberate in changes that are made, including writing code in a "linear" way when possible, such that merge conflicts are straightforward to resolve. However, the software development process itself cannot be linear, one must engage in test- and performance-driven development, designing with these aspects in mind from the start.

We both work in a computational group (the Computational Turbulent Reacting Flow Laboratory, CTRFL), and will seek to incorporate these lessons learned. In particular, our flow solvers have very few tests and and no continuous integration, such that many bugs propagate through commits and versions, even after they are found and resolved. While not relevant to this project, the course content in massively parallel architectures was also extremely relevant to our research.

This report presents the development of the Flow Instability Eulerian Flow Solver (`FIEFS`), a software repository to study flow instability. This existing code has been optimized to enhance user experience through a graphical user interface and a further developed data output and visualization scheme. It also includes improvements in the solver smoothing condition and has implemented a new instability cases for simulation. More figures and visualizations are available in the repository on `Github`. Further instructions on running the program are available in the `README` and specific documentation on each of the functions present in the code also available on the repository.

References

[1] J. Boerchers, S. Rzepka, and T. Fush. PSYCHo-I: Python Simulations Yielding Hydrodynamic Instabilities. https://github.com/johnboerchers/psycho-i, 2022.

[2] R.S. Cant and E. Mastorakos. *An Introduction to Turbulent Reacting Flows*. Imperial College Press, 2007.

[3] C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics: Fundamental and General Techniques*. Cambridge Aerospace Series. Springer-Verlag Berlin Heidelberg, 1998.

[4] T. Hynes and P. G. Tucker. *Course Lecture Slides*. Computational Fluid Dynamics Part IIB 4A2. University of Cambridge Department of Engineering, Michaelmas 2018.

[5] B. E. Launder and D. Spalding. The numerical computation of turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 3(2):269–289, 1974.

[6] T. Saad. *Turbulence Modeling For Beginners*. University of Tennessee Space Institute, 2011.

[7] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[8] P. G. Tucker. *Advanced Computational Fluid and Aerodynamics*. Cambridge Aerospace Series. Cambridge University Press, 2016.