APC 524 Final Project:

Implementing a Navier-Stokes Solver and Physics Informed Neural Network for Simulating Two-Dimensional Fluid Flow Around a Cylinder

Fairuz Ishraque[1], Joseph Lockwood[1], and Aaron Spaulding[2]

[1]Department of Geosciences
[2]Department of Civil and Environmental Engineering

Contents

1 Introduction

The Navier–Stokes (NS) equations describe fluid dynamics and have been applied to weather prediction, glacier dynamics, oceanography, thermal conduction, aircraft design, and architecture [5]. Despite their widespread use, analytic solutions exist only for a few constrained cases. As a result, the finite difference method (FDM) and finite element method (FEM) have become popular numerical approaches for obtaining approximate solutions [14]. These methods are computationally expensive, often requiring hundreds or thousands of core-hours to produce meaningful accurate results, and the largest simulations often require specialized hardware for effective scalability [8].

Recent advances in physics informed neural networks (PINNs) allow for high resolution and physically consistent approximations of the NS equations [7] [4] [6]. PINNs are supervised neural networks that take advantage of their capabilities as universal function approximators to incorporate model equations, such as partial differential equations, directly into the loss function during training [11]. This new loss term, known as the equation loss, is derived from the underlying physical system of equations accompanying the traditional mean square loss.

2 Overview

This project contains the following features:

1. An implementation of a modular Navier-Stokes solver using the Finite Difference Method.
   (a) A python class to define arbitrary environments and environmental conditions.
   (b) Modular boundary conditions allowing for many environment types to be investigated.
   (c) Modular and composable objects that allow for complex environments to be modeled and simulated.
2. A Physics Informed Neural Network that approximates a Navier-Stokes solver.
3. Unit tests implemented with "pytest"
4. Automated testing using GitHub Actions

3 Joseph's Contributions

3.1 A derivation and implementation of an NS solver to simulate the two-dimensional cylinder wake flow.

I implemented the initial finite-element Navier-Stokes (NS) solver [3] to simulate two-dimensional cylinder wake flow – the focus of which was on incompressible, viscous fluid flow [9]. The inclusion of viscosity in the model and the application of no-slip boundary conditions [13] is required to capture the nuanced behavior of real fluid flows often found in engineering applications. The presence of a square obstacle in the flow field allows us to investigate complex phenomena like separation, and vortex-shedding and wake formation [15, 1].

3.1.1 Navier-Stokes Equations

The NS equations, governing the fluid flow, are expressed as:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{1}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right), \tag{2}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{3}$$

where u and v are the fluid velocities in the x and y directions, respectively, p is the pressure, ρ is the fluid density, and ν is the kinematic viscosity.

3.1.2 Implementation

The solver was implemented in Python, utilizing NumPy for efficient array computations. It initializes the velocity and pressure fields, and defines a cylinder obstruction in the flow. The update functions for velocity and pressure discretize the NS equations using finite difference methods. The solver handles complex phenomena like vortex shedding, illustrated in the simulations around the cylinder. Regular plotting intervals offer visual insights into the evolving fluid dynamics.

3.1.3 Equations for Velocity and Pressure Update

The velocity and pressure updates incorporate advection, pressure gradients, and diffusion:

$$u_{next} = u - u\cdot dt\cdot\frac{\partial u}{\partial x} - v\cdot dt\cdot\frac{\partial u}{\partial y} - \frac{dt}{\rho}\frac{\partial p}{\partial x} + \nu\cdot dt\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{4}$$

$$v_{next} = v - u\cdot dt\cdot\frac{\partial v}{\partial x} - v\cdot dt\cdot\frac{\partial v}{\partial y} - \frac{dt}{\rho}\frac{\partial p}{\partial y} + \nu\cdot dt\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right). \tag{5}$$

The pressure update, derived from the Poisson equation to ensure incompressibility, is given by:

$$\begin{aligned} p_{next} = \frac{1}{2\cdot(dx^2+dy^2)}\Big(&(p_{i+1,j}+p_{i-1,j})\cdot dy^2 \\ &+ (p_{i,j+1}+p_{i,j-1})\cdot dx^2 \\ &- \rho\cdot((u_{i+1,j}-u_{i-1,j})/(2\cdot dx) + (v_{i,j+1}-v_{i,j-1})/(2\cdot dy))^2\cdot dx^2\cdot dy^2\Big). \end{aligned} \tag{6}$$

The simulation begins by initializing the velocity fields (representing fluid velocity in the x and y directions) and the pressure field. These initial conditions set the starting state of the fluid flow. The solver then discretizes the Navier-Stokes equations, which govern fluid motion, using finite difference methods in both time and space. To evolve the fluid dynamics over time, the solver employs update equations for velocity and pressure.

3.2 Unit testing

A comprehensive suite of unit tests has been implemented to ensure the robustness and reliability of the code. These tests encompass various aspects of the project, including the functionality of classes and methods, and the integrity of the environment setup. The testing approach was mainly implemented to test the reliability of each step in the solution process. The tests were designed to leverage the modular structure of the solver, enabling the examination of various functional components and critical variables within each of the utilized solver files. Consequently, a multitude of unit tests were developed to confirm the correct functioning of these distinct code segments.

Another key test is to ensure the Classes are properly initialized with default values. This test is vital for confirming that the simulation environment is set up correctly before any specific conditions or changes are applied. Additionally, we have also added tests targeted towards validating the internal mechanics of the Environment class, particularly ensuring that the methods responsible for updating matrices are functioning as expected. This is essential for the accuracy and reliability of the simulation's core computational algorithms.

These tests emphasize the importance of both verification and validation in software development. Verification ensures that the code meets set requirements and functions correctly, while validation confirms that these requirements make sense and serve the intended purpose. The tests implemented here serve as a testament to these principles, ensuring that the code not only works correctly but also fulfills its intended role effectively.

In line with the overarching principles of testing, the tests are designed to be adversarial, aiming to rigorously challenge and scrutinize the code rather than simply confirming its functionality. This approach ensures that any potential defects are identified and addressed, thereby enhancing the reliability and robustness of the code. The tests also serve as documentation, explicitly stating the expectations and requirements of the code, and they play a crucial role in growing confidence in its reliability among users.

3.3 Speed and efficiency

I integrated Python's cProfile [10] module as a step in identifying performance bottlenecks and speed of runs. This also allows for a concise comparison of the speed of the finite element NS solver against the PINN approximation. cProfile provides a detailed report on call frequency and duration of the code, which were pivotal in analyzing the efficiency. This integration was a key part of the process, allowing me to gather comprehensive performance data.

Subsequently, I executed the simulation with cProfile enabled, capturing essential performance metrics. This was critical in understanding how different segments of my code impacted the overall runtime, providing a clear overview of the simulation's performance landscape. For the analysis of the profiling data, I utilized SnakeViz [12], a sophisticated graphical viewer designed for Python profiling data. This analysis was integral in visualizing and decoding performance bottlenecks, highlighting areas where optimization was necessary (Fig. 1). The benefits of this process were multifaceted. Firstly, profiling pinpointed specific functions or methods that consumed substantial time, guiding me towards target areas for optimization. Moreover, it facilitated efficient allocation of computational resources by illuminating the most resource-intensive parts of the code, thereby aiding in more informed decisions regarding optimization.

With these insights, I embarked on optimizing and refining the code. By adjusting algorithms and refactoring, I was able to significantly improve the application's performance. This process not only enhanced the efficiency of the code but also its quality and scalability, elevating the overall standard of the software. Additionally, these optimizations led to time and cost efficiencies, particularly beneficial in environments where computational resources are charged. The data-driven approach provided by profiling enabled me to make informed decisions, concentrating my efforts on modifications that offered substantial performance improvements.

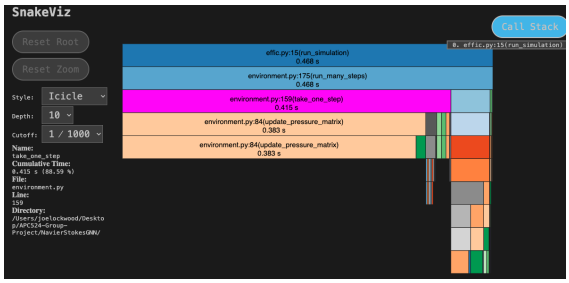


Figure 1: Sophisticated graphical viewer designed for Python profiling data using SnakeViz.

---

4 Aaron Spaulding's Contributions

4.1 An implementation of a Finite Difference Navier-Stokes solver to simulate the two-dimensional cylinder wake flow.

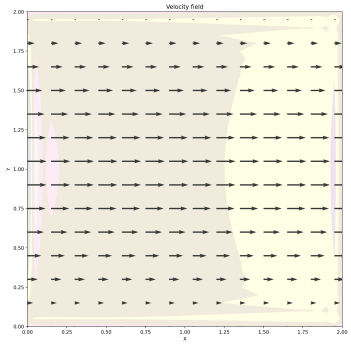
I implemented a solver for the Navier-Stokes equations using a Finite Difference(FD) method. This approach allows us to simulate fluid flows in two dimensions quickly and has been applied to many fluid related problems such as weather prediction, aerodynamics, and oceanography. I implemented this solver in python using the NumPy library for efficient array computations. The solver initializes the velocity and pressure fields and tracks changes and interactions between "parcels" of fluid interacting with each other. Each parcel is stationary and has a velocity and pressure associated with it that is updated at each time step to track the flux into and out of the parcel. This solver type enables efficient quick simulations at the cost of using fixed time steps and a fixed grid size. I implemented this solver following the equations derived by Joseph as well as work published from Barba et al.[2].

4.1.1 Environment Setup

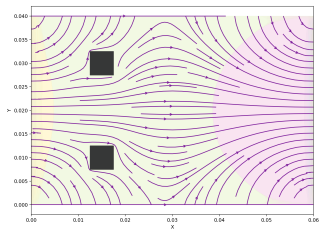
I designed the solver environment to be flexible and modular so users could easily define different size and resolution environments. The environment has a customizable grid, with adjustable resolution, time step, and fluid properties. This was implemented as a python "Environment" Class inside a module. The "Environment" class also includes automatic plotting routines that enable quick visualization of the fluid flow.

4.1.2 Boundary Conditions

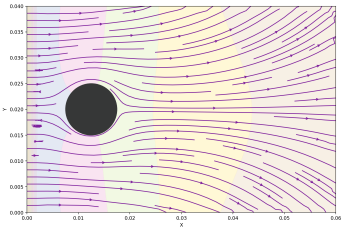
To extend this modularity I abstracted different types of boundary conditions common in fluid simulations. I implemented four different boundary conditions for each edge of the environment. Each of these boundary conditions is fully modular and can be mixed and matched for each simulation environment.



(a) Simulation of fluid flowing in a pipe. The top and bottom boundary conditions are set as no-slip conditions, while the left and right are set as periodic boundary conditions.



(b) Simulation of a fluid flowing around two boxes. Boundary conditions for all sides are set as fixed velocity conditions. The boundary conditions for the boxes are set as no-slip conditions are updated dynamically as each box is added to the environment.



(c) A simulation of fluid flow around a cylinder. Here the right, top, and bottom sides have no slip conditions while the left side has a fixed velocity boundary condition. The boundary conditions around the cylinder are automatically updated at simulation time.

Figure 2: The modular boundary conditions, customizable environment, and composable objects allows for easy simulation of complex environments with very different conditions and requirements.

1. **No-Slip Boundary Condition**: This boundary condition is used to simulate a solid boundary where fluid flows are zero in both the parallel and perpendicular directions to the boundary. This is seen on the inside of pipes, along buildings and objects, and against the ground.

2. **Fixed Velocity Boundary Condition**: This boundary condition is used to simulate a boundary where fluid flows are fixed in the parallel direction to the boundary. This could be used to simulate a fan, an inlent valve, or the top of a boundary layer where the fluid is moving at a constant velocity.

3. **Periodic Boundary Condition**: This boundary condition is used to simulate a boundary where fluid flows are periodic in the parallel direction to the boundary. This can be used to simulate repeating simulations such as a section of pipe where the input and output ends are similar.

4. **Free Slip Boundary Condition**: This boundary condition is used to simulate a boundary where fluid flows are zero in the perpendicular direction to the boundary. This can be used to simulate a boundary where the fluid is free to move in the parallel direction but cannot move in the perpendicular direction.

4.1.3 Objects in the Environment

Objects inside environments also interact with fluid flows and affect the velocity and pressure fields. To enable modular simulations, I also implemented an "Object" Class that can be used to place arbitrary objects in the environment. I implemented a "Rectangle" Class that inherits from the abstract "Object" Class that automatically manages boundary conditions of the added object and updates the velocity and pressure fields during simulation. I also implemented a "Cylinder" Class that also inherits from the abstract "Object" Class.

These can be combined to make complex simulations with multiple objects interacting with each other and the fluid flow.

4.1.4 Example Simulation Setup

```
from navier_stokes_fdm import Environment
from navier_stokes_fdm import Rectangle
import navier_stokes_fdm.boundary_condition as bc


U = 1  # m/s
dimension = 0.005

boundary_conditions = [
    bc.TopSideFixedVelocityBoundaryCondition(u_value=U, v_value=0),
    bc.BottomSideFixedVelocityBoundaryCondition(u_value=U, v_value=0),
    bc.LeftSideFixedVelocityBoundaryCondition(u_value=U, v_value=0),
    bc.RightSideFixedVelocityBoundaryCondition(u_value=U, v_value=0),
]


x1, y1 = 0.0125, (0.04 / 2) - (dimension / 2)
objects = [Rectangle(x1, y1, x1 + dimension, y1 + dimension)]


a = Environment(
    F=(1.0, 0.0),
    len_x=0.06,
    len_y=0.04,
    dt=0.00000015,
    dx=0.0001,
    boundary_conditions=boundary_conditions,
    objects=objects,
    rho=0.6125  # kg/m³
    nu=3e-5  # m²/s
)

a.run_many_steps(480)
a.plot_streamline_plot(title="", filepath="../Figures/box_example_streamline.png")
```

4.2 Unit testing

To further ensure code functionality I implemented unit tests using "pytest" for the boundary conditions. Each test creates an environment, applies a boundary condition, and checks to see if the boundary condition has been applied correctly.

4.2.1 Automated Unit Testing with GitHub Actions

To make sure that changes to the code do not break functionality, I implemented automated testing using GitHub Actions. I wrote a GitHub Action that runs the "pytest" unit tests on every pull request and commit to the repository.

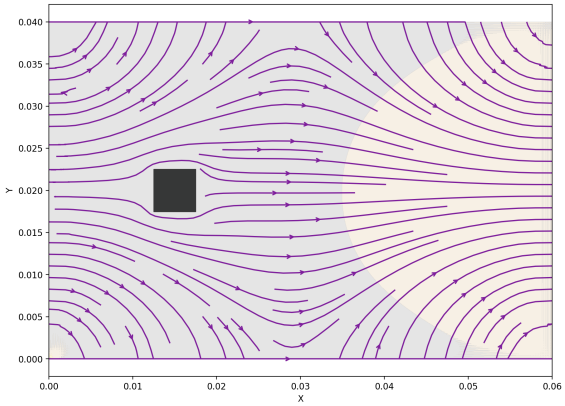
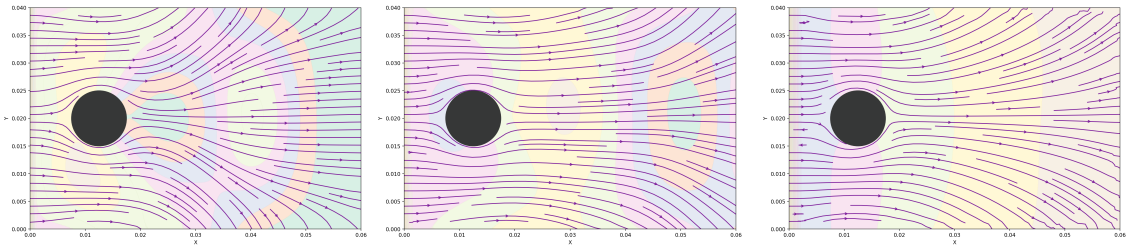


Figure 3: Example streamline plot of a fluid flow around a box. Each boundary is assigned a fixed velocity. Shading represents the pressure field with lighter colors indicating regions of lower pressure. Streamlines are shown in purple.

5 Simulation of a fluid flow around a cylinder

To simulate the fluid flow around a cylinder I initialized a simulation environment of $6cm$ by $4cm$ with a $5mm$ cylinder. I set ρ to be $0.6125\frac{kg}{m^3}$ and ν to be $3 * 10^{-5}\frac{m^2}{s}$. These values are physically plausible for air. The left side boundary condition was set to a fixed velocity of $1\frac{m}{s}$, and the top, right, and bottom boundaries were set as free slip conditions. The dt was set to $0.15\mu s$, and each step was a total of 30 time steps, or $4.5\mu s$. Every $4.5\mu s$ I generated and saved the streamline plot for analysis. Three selected frames are shown in Figure 4.

The simulation was stable until 10 time steps. After this point the simulation diverged. A longer simulation could be completed using a higher resolution grid, or by using smaller time steps.



(a) $9\mu s$ (b) $13.5\mu s$ (c) $45\mu s$

Figure 4: Three time steps of the FD simulation of fluid flow around a a cylinder. The pressure field is shown by the shading while streamlines are shown in purple.

---

6 Fairuz's Contributions

References

[1] John D. Anderson. *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill, 1995.

[2] Lorena A Barba and Gilbert F Forsyth. "CFD Python: the 12 steps to Navier-Stokes equations". In: *Journal of Open Source Education* 2.16 (2018), p. 21.

[3] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

[4] Mojtaba Baymani et al. "Artificial neural network method for solving the Navier–Stokes equations". In: *Neural Computing and Applications* 26 (2015), pp. 765–773.

[5] Alexandre Joel Chorin. "Numerical solution of the Navier-Stokes equations". In: *Mathematics of computation* 22.104 (1968), pp. 745–762.

[6] Hamidreza Eivazi et al. "Physics-informed neural networks for solving Reynolds-averaged Navier–Stokes equations". In: *Physics of Fluids* 34.7 (2022).

[7] Xiaowei Jin et al. "NSFnets (Navier-Stokes flow nets): Physics-informed neural networks for the incompressible Navier-Stokes equations". In: *Journal of Computational Physics* 426 (2021), p. 109951.

[8] John Michalakes et al. "WRF nature run". In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. 2007, pp. 1–6.

[9] Stephen B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[10] Python Software Foundation. *cProfile — Python Profiler*. Accessed: December 2023.

[11] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (Feb. 2019), pp. 686–707. DOI: 10.1016/j.jcp.2018.10.045. URL: https://doi.org/10.1016/j.jcp.2018.10.045.

[12] *SnakeViz*. https://jiffyclub.github.io/snakeviz/. Accessed: December 2023.

[13] Frank M. White. *Viscous Fluid Flow*. 3rd. McGraw-Hill Higher Education, 2006.

[14] Jonathan Whiteley. *Finite Element Methods*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-49971-0. URL: https://doi.org/10.1007/978-3-319-49971-0.

[15] M. M. Zdravkovich. *Flow around Circular Cylinders: A Comprehensive Guide through Flow Phenomena, Experiments, Applications, Mathematical Models, and Computer Simulations*. Vol. 1. Oxford University Press, 1997.