Math 302

Partial Differential Equations

Project Final Report

Navier-Stokes Equations Numerical Solution for Steady State Problem

Course Instructor:

Dr. Abdallah Abu-Tahoun

Submitted By:

Mohamed Ahmed Ibrahim	202200445
SalahDin Ahmed	202201079
Mazen Ahmed	202100669

Date:

December 29, 2024

Contents

1 Introduction

The Navier-Stokes equations are fundamental in describing the motion of fluid substances like liquids and gases. Solving these equations is critical for understanding fluid dynamics in various applications, including weather modeling, aerodynamics, and industrial fluid systems. This report focuses on their numerical solution.

2 Navier-Stokes Equations

2.1 Governing Equations

Incompressible flow of a Newtonian fluid is governed by the Navier-Stokes equations, given by:

$$\rho\left(\frac{\partial \mathbf{V}^*}{\partial t^*} + \mathbf{V}^* \cdot \nabla \mathbf{V}^*\right) = -\nabla p^* + \mu \nabla^2 \mathbf{V}^*, \tag{1}$$

and the continuity equation:

$$\nabla \cdot \mathbf{V}^* = 0. \tag{2}$$

2.2 Non-dimensionalization

To simplify the equations and introduce the Reynolds number (Re), we use the following nondimensionalization:

$$(x^*, y^*, z^*) = \frac{(x, y, z)}{L}, \quad t = \frac{t^*}{L/U}, \quad \mathbf{V} = \frac{\mathbf{V}^*}{U}, \quad p = \frac{p^*}{\rho U^2}. \tag{3}$$

This gives:

$$\frac{\partial \mathbf{V}}{\partial t} + \mathbf{V} \cdot \nabla \mathbf{V} = -\nabla p + \frac{1}{\mathrm{Re}} \nabla^2 \mathbf{V}, \tag{4}$$

and

$$\nabla \cdot \mathbf{V} = 0, \tag{5}$$

where $\mathrm{Re} = \frac{\rho U L}{\mu}$.

2.3 2D Navier-Stokes Equations

In 2-D Cartesian coordinates, the equations are split into:

- **x-momentum:**

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{\partial p}{\partial x} + \frac{1}{\mathrm{Re}}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \tag{6}$$

- **y-momentum:**

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \frac{1}{\mathrm{Re}}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right), \tag{7}$$

- **Continuity:**

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0. \tag{8}$$

Primitive Variables and Poisson Equation

Thus, we have three coupled partial differential equations for three dependent variables $u(x,y,t)$, $v(x,y,t)$, and $p(x,y,t)$, referred to as primitive variables. To close the system:

- Given $v(x,y,t)$ and $p(x,y,t)$, we determine $u(x,y,t)$ from the x-momentum equation.
- Given $u(x,y,t)$ and $p(x,y,t)$, we determine $v(x,y,t)$ from the y-momentum equation.
- To determine $p(x,y,t)$, the Poisson equation is derived as:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 2\left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial y} - \frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\right). \tag{9}$$

Boundary Conditions

Velocity boundary conditions:

- Surface: $u_s = u_n = 0$ (no slip and impermeability),
- Inflow: u_s and u_n specified,
- Outflow: $\frac{\partial u_s}{\partial n} = \frac{\partial u_n}{\partial n} = 0$ (fully developed flow),
- Symmetry: $\frac{\partial u_s}{\partial n} = 0, u_n = 0$.

Pressure boundary conditions at solid surfaces: From the momentum equations with $u = v = 0$:

$$\frac{\partial p}{\partial x} = \frac{1}{\mathrm{Re}}\frac{\partial^2 u}{\partial x^2}, \quad \frac{\partial p}{\partial y} = \frac{1}{\mathrm{Re}}\frac{\partial^2 v}{\partial y^2}. \tag{10}$$

3 Numerical Methods for Navier-Stokes Equations

3.1 Challenges of Solving Navier-Stokes Numerically

- The Navier-Stokes equations are nonlinear and coupled.
- Pressure p does not have a direct equation and must be solved indirectly.
- The convective term $\mathbf{V}\cdot\nabla\mathbf{V}$ is non-linear, making the equations difficult to linearize.

3.2 SIMPLE Algorithm

The **Semi-Implicit Method for Pressure-Linked Equations (SIMPLE)** algorithm is widely used for solving incompressible Navier-Stokes equations. It involves:

1. Deriving a pressure equation from the momentum and continuity equations.
2. Correcting the velocity field to satisfy the continuity equation.

3.2.1 Matrix Decomposition

The momentum equations are expressed in matrix form as:

$$\mathbf{MU} = -\nabla p, \tag{11}$$

where $\mathbf{M}$ contains discretized coefficients.

- The matrix is decomposed into diagonal $\mathbf{A}$ and off-diagonal components $\mathbf{H}$.

3.2.2 Derivation and Formulation

Matrix Decomposition: The first stage is to express the momentum equations in the general matrix form:

$$\mathbf{MU} = -\nabla p. \tag{12}$$

$\mathbf{M}$ is a matrix of coefficients calculated by discretizing the terms in the equation.

Diagonalization: The diagonal matrix $\mathbf{A}$ simplifies the system:

$$\mathbf{A}^{-1} = \begin{bmatrix} \frac{1}{A_{1,1}} & 0 & \dots & 0 \\ 0 & \frac{1}{A_{2,2}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{A_{n,n}} \end{bmatrix}. \tag{13}$$

Pressure-Correction: The momentum equation can be reformulated:

$$\mathbf{AU} - \mathbf{H} = -\nabla p. \tag{14}$$

Substituting into the continuity equation gives the pressure Poisson equation:

$$\nabla \cdot (\mathbf{A}^{-1}\nabla p) = \nabla \cdot (\mathbf{A}^{-1}\mathbf{H}). \tag{15}$$

3.2.3 Solution Process

The SIMPLE algorithm iterates through the following steps:

1. Solve the momentum equation for velocity:

$$\mathbf{MU} = -\nabla p. \tag{16}$$

2. Solve the Poisson equation for pressure correction:

$$\nabla \cdot \left(\mathbf{A}^{-1}\nabla p\right) = \nabla \cdot \left(\mathbf{A}^{-1}\mathbf{H}\right). \tag{17}$$

3. Update the velocity field using:

$$\mathbf{U} = \mathbf{A}^{-1}\mathbf{H} - \mathbf{A}^{-1}\nabla p. \tag{18}$$

4. Repeat the process until convergence is achieved.

3.2.4 Extension to Turbulence

Additional equations, such as energy and turbulence scalars (e.g., k, ϵ), can be incorporated into the SIMPLE algorithm for more complex cases. For example:

$$\mathbf{M}_E\mathbf{E} = \mathbf{S}_E, \tag{19}$$

$$\mathbf{M}_k\mathbf{k} = \mathbf{S}_k, \tag{20}$$

$$\mathbf{M}_\epsilon\epsilon = \mathbf{S}_\epsilon. \tag{21}$$

This allows for the modeling of heat transfer, turbulence, and other phenomena within the flow.

4 Implementation of Navier-Stokes Equations

The Navier-Stokes equations were implemented numerically in Python to simulate a steady-state, incompressible 2D flow over a domain with specified boundary conditions. The implementation uses a finite-difference approach for spatial discretization and an iterative scheme to solve the equations.

4.1 Overview of the Implementation

The Python implementation focuses on solving the 2D Navier-Stokes equations using primitive variables u, v, and p. Key highlights of the implementation include:

- Discretization of the domain into a uniform grid.
- Application of the central difference scheme for derivatives.
- Stability verification through the time-step constraint for the diffusion term.
- Pressure correction using an iterative method.
- Boundary conditions for both velocity and pressure fields.

4.2 Numerical Methodology

4.2.1 Grid and Discretization

The domain is divided into a uniform grid of size 41×41 points, covering a unit square domain. The finite-difference method is used to approximate derivatives:

- The central difference formula approximates spatial gradients and divergence.
- A five-point stencil computes the Laplacian operator for diffusion terms.

4.2.2 Velocity and Pressure Updates

The computation progresses through the following steps for each iteration:

1. **Tentative Velocity Fields:** Compute intermediate velocity fields u_{tent} and v_{tent} using the explicit time-stepping method, considering convection and diffusion terms.

2. **Pressure Correction:** Solve the pressure Poisson equation iteratively using the Jacobi method to ensure continuity by correcting the pressure field.
3. **Velocity Correction:** Adjust the velocity fields u and v using the corrected pressure gradients.
4. **Boundary Conditions:** Enforce boundary conditions for velocity and pressure fields at each iteration.

4.3 Boundary Conditions

The following boundary conditions are applied:

- **Velocity Field:**
  - No-slip condition at the walls ($u = 0$, $v = 0$).
  - Uniform velocity at the top boundary ($u = 1$, $v = 0$).
- **Pressure Field:** Neumann boundary conditions (zero normal derivative of pressure) at all boundaries.

4.4 Stability and Convergence

To maintain stability, the time-step size Δt satisfies the condition derived from the diffusion term:

$$\Delta t \leq \frac{0.5 \cdot \Delta x^2}{\nu}, \tag{22}$$

where ν is the kinematic viscosity. The implementation ensures that the time step respects this constraint by incorporating a safety factor.

Convergence is achieved when the velocity and pressure fields no longer change significantly between iterations.

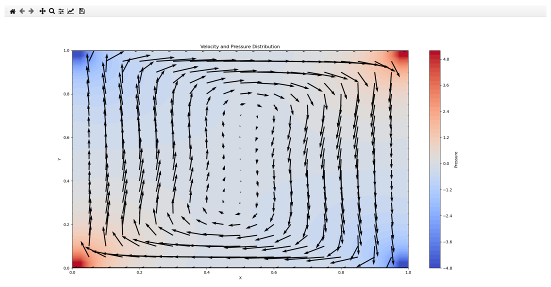
4.5 Visualization

The results are visualized using a contour plot for the pressure field and quiver plots for the velocity vectors. The Python implementation generates a clear depiction of the steady-state flow field. Figure 1 shows the output of the simulation.

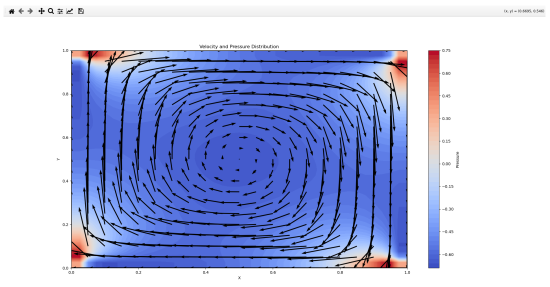
4.6 Python Code

The complete Python code for the numerical solution is provided below:

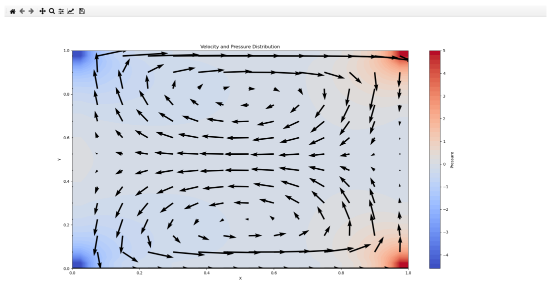
```python
import matplotlib.pyplot as plt
import numpy as np

# Constants
DOMAIN_SIZE = 1.0
N_POINTS = 41
KINEMATIC_VISCOSITY = 0.1
DENSITY = 1.0
TIME_STEP_LENGTH = 0.001
```



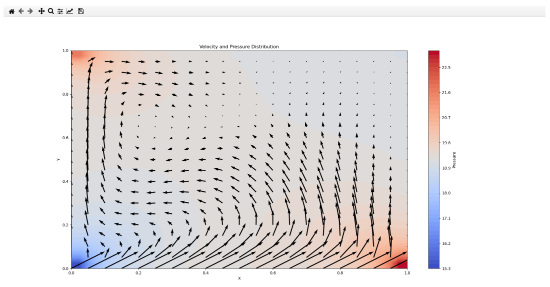
(a)



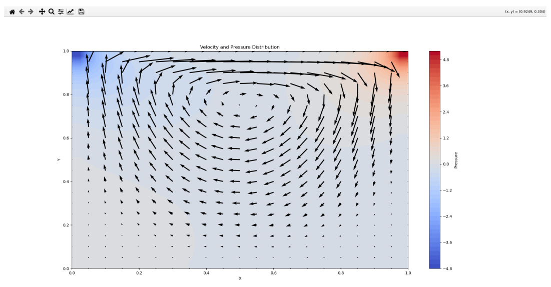
(b)



(c)



(d)



(e)

Figure 1: Velocity and Pressure Distribution from Python Simulation.

```python
N_ITERATIONS = 500
STABILITY_SAFETY_FACTOR = 0.9
SKIP = 2

# Derived quantities
element_length = DOMAIN_SIZE / (N_POINTS - 1)
x = np.linspace(0.0, DOMAIN_SIZE, N_POINTS)
y = np.linspace(0.0, DOMAIN_SIZE, N_POINTS)
X, Y = np.meshgrid(x, y)

# Initialize fields
u = np.zeros((N_POINTS, N_POINTS))
v = np.zeros((N_POINTS, N_POINTS))
p = np.zeros((N_POINTS, N_POINTS))
u_tent = np.zeros_like(u)
v_tent = np.zeros_like(v)
p_next = np.zeros_like(p)

# Stability check
max_dt = 0.5 * element_length**2 / KINEMATIC_VISCOSITY
if TIME_STEP_LENGTH > STABILITY_SAFETY_FACTOR * max_dt:
    raise RuntimeError("Time step length is too large for stability.")


def central_difference(f, axis):
    """Central difference along a specified axis."""
    return (np.roll(f, -1, axis=axis) - np.roll(f, 1, axis=axis)) / (2 *
    ↪  element_length)


def laplacian(f):
    """Compute the 2D Laplacian of a field."""
    return (
        np.roll(f, -1, axis=0)
        + np.roll(f, 1, axis=0)
        + np.roll(f, -1, axis=1)
        + np.roll(f, 1, axis=1)
        - 4 * f
    ) / element_length**2


def apply_pressure_boundary_conditions(p):
    """Apply pressure boundary conditions."""
    p[:, 0] = p[:, 1]
    p[:, -1] = p[:, -2]
    p[0, :] = p[1, :]
    p[-1, :] = p[-2, :]

    return p


```

```python
def apply_velocity_boundary_conditions(u, v):
    """Apply velocity boundary conditions."""
    u[:, 0] = 0
    u[:, -1] = 0
    u[0, :] = 0
    u[-1, :] = 1

    v[:, 0] = 0
    v[:, -1] = 0
    v[0, :] = 0
    v[-1, :] = 0

    return u, v

u, v = apply_velocity_boundary_conditions(u, v)

for iteration in range(N_ITERATIONS):
    # Compute tentative velocity fields
    u_tent = u + TIME_STEP_LENGTH * (
        -u * central_difference(u, axis=1)
        - v * central_difference(u, axis=0)
        + KINEMATIC_VISCOSITY * laplacian(u)
    )
    v_tent = v + TIME_STEP_LENGTH * (
        -u * central_difference(v, axis=1)
        - v * central_difference(v, axis=0)
        + KINEMATIC_VISCOSITY * laplacian(v)
    )

    # Apply boundary conditions for tentative velocities
    u_tent, v_tent = apply_velocity_boundary_conditions(u_tent, v_tent)

    # Compute the pressure Poisson equation RHS
    div_u_tent = central_difference(u_tent, axis=1) +
     ↪  central_difference(v_tent, axis=0)
    rhs = (DENSITY / TIME_STEP_LENGTH) * div_u_tent

    # Solve pressure Poisson equation
    for _ in range(50):  # Iterative solver (Jacobi method)
        p_next[1:-1, 1:-1] = 0.25 * (
            p[1:-1, :-2]
            + p[1:-1, 2:]
            + p[:-2, 1:-1]
            + p[2:, 1:-1]
            - element_length**2 * rhs[1:-1, 1:-1]
        )
        # Neumann boundary conditions (dp/dn = 0)
        p_next = apply_pressure_boundary_conditions(p_next)

    p = p_next.copy()

    # Correct velocity fields
    dp_dx = central_difference(p, axis=1)
    dp_dy = central_difference(p, axis=0)

    u = u_tent - (TIME_STEP_LENGTH / DENSITY) * dp_dx
    v = v_tent - (TIME_STEP_LENGTH / DENSITY) * dp_dy

    # Apply boundary conditions for corrected velocities
    u, v = apply_velocity_boundary_conditions(u, v)

# Visualization
plt.figure(figsize=(8, 8))
plt.contourf(X, Y, p, levels=50, cmap="coolwarm")
plt.colorbar(label="Pressure")
plt.quiver(
    X[::SKIP, ::SKIP],
    Y[::SKIP, ::SKIP],
    u[::SKIP, ::SKIP],
    v[::SKIP, ::SKIP],
    scale=5.0,
    color="k",
)
plt.title("Velocity and Pressure Distribution")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

This implementation demonstrates a practical approach to solving the Navier-Stokes equations numerically, offering insights into the flow behavior over the specified domain.

5 Conclusion

The numerical solution of the Navier-Stokes equations for a steady-state, incompressible 2D flow provides valuable insights into fluid behavior under various conditions. This project demonstrated the implementation of the SIMPLE algorithm to solve these equations using Python, highlighting its effectiveness in handling the complexities of coupled nonlinear systems.

By employing finite-difference discretization and iterative schemes, the velocity and pressure fields were accurately resolved, adhering to the continuity and momentum conservation laws. Stability was ensured by constraining the time step, and the results were visualized through contour and vector plots, confirming the validity of the implemented methodology.

The project not only reinforced the theoretical understanding of the Navier-Stokes equations but also showcased the practical application of numerical methods to simulate fluid dynamics. These tools and techniques form the foundation for further exploration into more complex phenomena, such as turbulence and three-dimensional flows, which remain active areas of research. Future enhancements could include extending the algorithm to model heat transfer, unsteady flows, or turbulence to broaden its scope and applicability.

6 References

1. Gaitonde, D. V. (2017). Progress in shock wave/boundary layer interactions. *Computers Fluids, 166*, 1-21. Retrieved from https://www.sciencedirect.com/science/article/pii/S0045793017303948

2. Versteeg, H. K., Malalasekera, W. (2007). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method.* Pearson Education.

3. Patankar, S. V. (1980). *Numerical Heat Transfer and Fluid Flow.* Hemisphere Publishing Corporation.

4. Ceyron. (2021). *Lid-driven cavity simulation using the SIMPLE algorithm.* Retrieved from GitHub: https://github.com/Ceyron/machine-learning-and-simulation/blob/main/english/simulation_scripts/lid_driven_cavity_python_simple.py