2D Numerical Simulation of Flow Past a Circular Cylinder: Solver Development, Validation, and Wake Analysis

Hamza Hussain

Independent Technical Report *June 2, 2026*

Abstract

A two-dimensional incompressible Navier–Stokes solver is developed and validated against the experimental drag coefficient correlation of Schlichting [1] for flow past a circular cylinder over the range $Re \in [10^{-1}, 10^{6}]$. The cylinder geometry is represented on a fixed Cartesian grid using a mask-based immersed-boundary treatment, in which a Boolean solid indicator identifies the cylinder and the no-slip condition is imposed by zeroing the velocity inside the body at each time step. Pressure–velocity coupling is handled through Chorin's fractional-step projection method [2], and convective derivatives are discretised with a second-order directional upwind-biased scheme. An adaptive time-stepping strategy simultaneously satisfies the hyperbolic CFL and parabolic diffusion stability constraints at each step. The solver reproduces the experimental drag coefficient to acceptable agreement in the low-Reynolds-number regime and captures the expected transition from steady separated wake structure to alternating vortex shedding at higher Reynolds number. Discrepancies at higher Re are attributed to the inherently two-dimensional formulation, the Cartesian representation of the curved wall, and insufficient near-wall resolution, which preclude reproduction of the drag crisis near $Re \approx 9 \times 10^5$. These results establish a validated baseline for planned active flow control studies involving surface jet blowing.

1. Introduction

Relative motion between a solid body and a fluid occurs in a wide range of flows. Examples include the motion of aircraft through air, submarine bodies through water, wind passing bridge cables and towers, and flow around bluff structural members. Although practical configurations are often geometrically complex, many of the essential physical mechanisms can be studied using a much simpler canonical geometry. The circular cylinder in cross-flow is therefore a useful benchmark: it removes unnecessary geometric complexity while retaining separation, recirculation, vortex shedding, pressure drag, and Reynolds-number-dependent wake behaviour.

In the idealised two-dimensional problem, a cylinder of diameter D is placed with its axis normal to a uniform incoming flow of speed U. If the cylinder is sufficiently long compared with its diameter, and if the end effects are negligible, the central portion of the flow may be treated as approximately two-dimensional. Similarly, if the external boundaries are sufficiently far from the cylinder, their influence on the near wake is reduced. This makes the problem suitable for code development because the geometry is simple, the governing physics remain non-trivial, and extensive experimental data are available for comparison.

Flow past a bluff body is one of the canonical problems of fluid dynamics, the circular cylinder in cross-flow has served as a benchmark for computational methods for decades, owing to its geometric simplicity and the wealth of experimental data available for validation. At low Reynolds number, the wake may remain steady and attached or form a steady recirculation region behind the body. As inertial effects increase, the separated shear layers become unstable and roll into alternating vortices, forming the classical Kármán wake.

The drag coefficient C_D exhibits qualitatively distinct behaviour across several decades of Re. At very low Re, viscous forces dominate and $C_D \sim Re^{-1}$ in the Stokes regime. With increasing Re, the wake transitions through steady attached flow, laminar separation, steady recirculation, and periodic vortex shedding, culminating in the drag crisis near $Re \approx 9 \times 10^5$, where boundary-layer transition delays separation and C_D drops sharply.

Numerical simulation of this flow presents several challenges: adequate resolution of thin boundary layers, control of numerical diffusion in the convective terms, accurate pressure–velocity coupling in the incompressible limit, and the representation of a curved solid boundary on a structured Cartesian grid. The present work addresses these within a finite-difference framework and documents the solver's capabilities and limitations through comparison with experiment and qualitative wake visualisation.

2. Mathematical Formulation

2.1. Incompressible Navier–Stokes Equations

The motion of a viscous, incompressible, Newtonian fluid is governed by the momentum equation

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

and the incompressibility constraint

$$\nabla \cdot \mathbf{V} = 0, \tag{2}$$

where $\mathbf{V} = u\,\hat{\mathbf{i}} + v\,\hat{\mathbf{j}}$ is the velocity field, p the static pressure, ρ the density, and $\nu = \mu/\rho$ the kinematic viscosity. The non-linear convective term $(\mathbf{V} \cdot \nabla)\mathbf{V}$ transports momentum by the bulk flow; the Laplacian term $\nu \nabla^2 \mathbf{V}$ represents viscous diffusion, namely the redistribution

of momentum from regions of high to low velocity due to internal friction, strongest near solid walls and in regions of sharp velocity gradients. Being simultaneously first-order in time and second-order in space, and non-linear in the convective term, the equations are troublesome and admit closed-form analytical solutions only for a restricted class of highly symmetric flows.

2.2. Physical Interpretation of the Continuity Constraint

The kinematic significance of (2) is clarified by considering a Lagrangian fluid parcel of instantaneous volume $\delta\bar{V}$ moving with the flow. Following the material-volume interpretation commonly presented in Anderson's introductory treatment of computational fluid dynamics [3], the divergence of velocity may be related to the fractional rate of change of a fluid element's volume:

$$\nabla \cdot \mathbf{V} = \frac{1}{\delta\bar{V}} \frac{D\delta\bar{V}}{Dt}. \tag{3}$$

Thus, $\nabla\cdot\mathbf{V}$ is the fractional rate of volumetric expansion of the parcel per unit volume. Equation (2) therefore states that no fluid parcel may change its volume. The parcel may translate, rotate, and deform in shape, but it cannot undergo net volumetric expansion or compression.

Any isotropic deformation, meaning volumetric expansion or compression, requires equal normal stresses acting in all directions simultaneously. In the Navier–Stokes equations, the scalar pressure field is the mechanism capable of exerting such isotropic stress, since it acts uniformly on all faces of a fluid element. The incompressibility constraint is therefore a statement that the pressure field must adjust throughout the domain to prevent net volumetric change in each material parcel. This global coupling between velocity divergence and pressure is the fundamental source of the pressure–velocity coupling problem in incompressible flow solvers, addressed in Section 3.2.

2.3. Reynolds Number

The dimensionless Reynolds number

$$Re = \frac{UD}{\nu} \tag{4}$$

characterises the ratio of inertial to viscous forces, with U the free-stream speed and $D = 2R$ the cylinder diameter. In the present study, U is varied to span the target Reynolds numbers while ρ and ν are held fixed.

3. Numerical Methodology

3.1. Immersed Boundary Treatment of the Cylinder

The approach adopted here is best described as a mask-based immersed-boundary treatment. The solid body is represented by a Boolean mask function defined on the background Cartesian grid. This permits the cylinder to be imposed directly on the structured mesh used by the finite-difference solver.

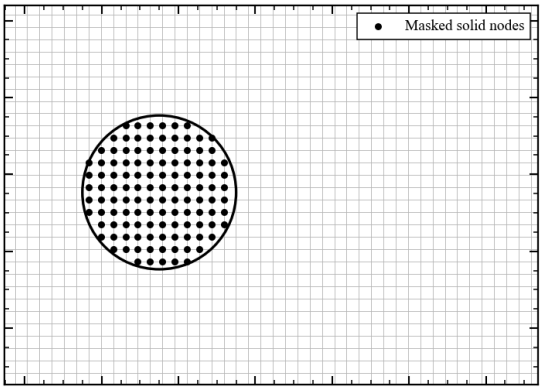


Figure 1: Cartesian mesh representation of the immersed circular cylinder. The black points indicate masked solid nodes where the no-slip condition is imposed by setting $u = v = 0$. The circular outline is shown for reference, illustrating the staircase-type approximation introduced by representing a curved boundary on a Cartesian grid.

In the present implementation, solid cells are identified by the mask

$$M_{i,j} = \begin{cases} 1 & \text{if } (x_{i,j} - x_c)^2 + (y_{i,j} - y_c)^2 \le R^2, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

The resulting masked-cylinder representation is shown in Figure 1. The circular boundary is superimposed on the underlying Cartesian grid, while the black nodes denote the solid cells inside the cylinder. These nodes are not treated as fluid points during the velocity update; instead, the velocity inside the mask is reset to zero when the boundary conditions are applied.

The no-slip condition is enforced sharply at each time step by explicitly setting $u = v = 0$ at all cells where $M_{i,j} = 1$. The velocity field is thereby extended into the solid region as a zero-valued field, consistent with a stationary rigid cylinder. This treatment is computationally simple and avoids the need for body-fitted meshing, but the curved wall is represented approximately on a Cartesian grid. The mesh therefore introduces a staircase-type geometric error at the cylinder surface, which is one of the limitations of the present solver.

The pressure boundary condition on the cylinder is approximated using a zero-normal-gradient condition. Rather than solving a modified Poisson equation near the curved surface, the pressure inside each masked cell is assigned the value of a neighbouring fluid cell. The domain is partitioned into four directional sub-masks corresponding to right, left, top, and bottom regions of the circular mask, and the relevant source indices are pre-computed before the time loop.

3.2. Chorin's Fractional-Step Projection Method

Pressure–velocity coupling is handled through Chorin's projection method [2], which decomposes each time step $n \to n + 1$ into three sub-steps.

```python
def zeropressuregrad_fast(p,mask_right,mask_left,
                          mask_top,mask_bottom,
                          right_src,left_src,
                          top_src,bottom_src):
    p[mask_right]  = p[right_src]
    p[mask_left]   = p[left_src]
    p[mask_top]    = p[top_src]
    p[mask_bottom] = p[bottom_src]
    return p
```

Listing 1: Masked-cylinder pressure copying used to approximate a zero-normal-gradient wall pressure condition.

```python
def buildup(rho, dt, u, v, dx, dy):
    b = np.zeros((ny, nx))
    b[2:-2, 2:-2] = (rho / dt) * (
        (u[2:-2, 3:-1] - u[2:-2, 1:-3]) / (2 * dx)
      + (v[3:-1, 2:-2] - v[1:-3, 2:-2]) / (2 * dy)
    )
    return b
```

Listing 2: Assembly of the pressure Poisson right-hand side from the divergence of the intermediate velocity field.

3.2.1. Intermediate velocity advance

An intermediate, generally non-solenoidal velocity field $\mathbf{V}^*$ is obtained by advancing the momentum equation with the pressure gradient omitted:

$$\frac{\mathbf{V}^* - \mathbf{V}^n}{\Delta t} = -(\mathbf{V}^n \cdot \nabla)\mathbf{V}^n + \nu \nabla^2 \mathbf{V}^n. \tag{6}$$

3.2.2. Pressure Poisson equation

Imposing $\nabla \cdot \mathbf{V}^{n+1} = 0$ on the velocity correction $\mathbf{V}^{n+1} = \mathbf{V}^* - (\Delta t/\rho)\nabla p^{n+1}$ yields the pressure Poisson equation:

$$\nabla^2 p^{n+1} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{V}^*. \tag{7}$$

This is solved iteratively until the relative pressure update satisfies $\|\delta p\|/\|p\| \leq 10^{-5}$, with the immersed wall pressure condition reapplied during the Poisson iteration. The right-hand side is assembled as shown in Listing 2.

3.2.3. Dimensional consistency of the velocity correction

The velocity correction

$$\mathbf{V}^{n+1} = \mathbf{V}^* - \frac{\Delta t}{\rho} \nabla p^{n+1} \tag{8}$$

requires $(\Delta t/\rho)\nabla p$ to carry units of velocity. Verifying:

$$\left[\frac{\Delta t}{\rho}\,\frac{\partial p}{\partial x}\right] = \frac{[\mathrm{s}]}{[\mathrm{kg\,m^{-3}}]} \cdot \frac{[\mathrm{kg\,m^{-1}\,s^{-2}}]}{[\mathrm{m}]} = \mathrm{m\,s^{-1}}, \tag{9}$$

confirming dimensional consistency. In the discretised form the x-correction reads

$$u_{i,j}^{n+1} = u_{i,j}^* - \frac{\Delta t}{2\rho\,\Delta x}\left(p_{i,j+1} - p_{i,j-1}\right). \tag{10}$$

```python
du_dx_backward = (
    3 * un[2:-2, 2:-2]
  - 4 * un[2:-2, 1:-3]
  +     un[2:-2, 0:-4]
) / (2 * dx)

du_dx_forward = (
   -3 * un[2:-2, 2:-2]
  + 4 * un[2:-2, 3:-1]
  -     un[2:-2, 4:]
) / (2 * dx)

du_dx = np.where(un[2:-2, 2:-2] < 0,
                 du_dx_forward,
                 du_dx_backward)
```

Listing 3: Second-order directional upwind convective derivative used in the solver.

3.3. Spatial Discretisation: Second-Order Directional Upwind Scheme

3.3.1. Deficiency of first-order upwind differencing

A first-order upwind scheme introduces numerical viscosity $\nu_{\mathrm{num}} \approx \frac{1}{2}U\Delta x$, yielding an effective Reynolds number

$$Re_{\mathrm{eff}} = \frac{UL}{\nu + \frac{1}{2}U\Delta x}. \tag{11}$$

As U is increased, $\nu_{\mathrm{num}} \gg \nu$ and Re_{eff} asymptotes to a mesh-dependent ceiling:

$$Re_{\mathrm{eff}} \approx \frac{UL}{\frac{1}{2}U\Delta x} = \frac{2L}{\Delta x}. \tag{12}$$

The velocity U cancels: regardless of the physical Reynolds number, the effective Reynolds number is bounded by the mesh alone. Reynolds-number-dependent wake structure is then suppressed by artificial diffusion before it can fully develop, rendering a first-order scheme unsuitable for the present study, which requires testing of higher Reynolds-numbers.

3.3.2. Second-order directional upwind stencil

The present solver employs three-point one-sided stencils selected locally by the sign of the advecting velocity. For the x-derivative of u:

$$\left.\frac{\partial u}{\partial x}\right|_{\mathrm{bwd}} = \frac{3u_{i,j} - 4u_{i,j-1} + u_{i,j-2}}{2\Delta x} + \mathcal{O}(\Delta x^2), \tag{13}$$

$$\left.\frac{\partial u}{\partial x}\right|_{\mathrm{fwd}} = \frac{-3u_{i,j} + 4u_{i,j+1} - u_{i,j+2}}{2\Delta x} + \mathcal{O}(\Delta x^2). \tag{14}$$

The backward stencil (13) is used where $u > 0$ and the forward stencil (14) where $u < 0$, drawing information from the physically upstream direction. Although the stencil uses three points, its truncation error is second order. The reduced artificial diffusion allows sharper wake structures to remain visible, but the lower numerical damping demands a smaller Courant number for stability. Listing 3 shows the implementation.

3.4. Adaptive Time-Stepping

At each time step, Δt is set:

$$\Delta t_{\text{adv}} = \sigma\left(\frac{|\mathbf{V}|^x_{\max}}{\Delta x} + \frac{|\mathbf{V}|^y_{\max}}{\Delta y}\right)^{-1}, \tag{15}$$

$$\Delta t_{\text{diff}} = \frac{\sigma}{2}\left[\nu\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)\right]^{-1}, \tag{16}$$

with $\sigma = 0.3$ and $\Delta t = \min(\Delta t_{\text{adv}}, \Delta t_{\text{diff}})$. The diffusion limit $d = \nu\Delta t/(\Delta x)^2 \leq 0.5$ follows from Von Neumann stability analysis of the heat equation. Similarly, the advective equation is found by analysing the inviscid burger's equation. σ is the safety factor, which is set low to ensure dispersion does not cause the catastrophic blow-up of a solution.

3.5. Boundary Conditions

At the inlet ($x = 0$), a uniform Dirichlet condition $u = U$, $v = 0$ is prescribed on the physical boundary and the adjacent ghost layer. At the outlet ($x = L_x$), a zero-gradient Neumann condition is applied to both velocity components with $p = 0$. The top and bottom boundaries are treated as open Neumann boundaries. No-slip on the cylinder is enforced through the immersed solid mask (Section 3.1). The pressure Neumann condition on the cylinder is approximated by directional copying from neighbouring fluid cells.

3.6. Drag Coefficient Calculation

Pressure is sampled at $N_\theta = 80$ uniformly spaced angular positions θ_k, one grid spacing outside the cylinder surface. The pressure-drag force and drag coefficient are

$$F_D = -R\sum_{k=1}^{N_\theta} p(\theta_k)\cos\theta_k\,\Delta\theta, \qquad C_D = \frac{F_D}{\frac{1}{2}\rho U^2 D}, \tag{17}$$

where $\Delta\theta = 2\pi/N_\theta$. Only pressure drag is included; viscous skin-friction is not computed.

4. Nomenclature

C_D	drag coefficient
D	cylinder diameter, $2R$
d	diffusion number, $\nu\Delta t/(\Delta x)^2$
F_D	drag force (N)
$M_{i,j}$	immersed boundary mask function
N_θ	number of surface pressure probes
p	static pressure (Pa)
R	cylinder radius (m)
Re	Reynolds number, UD/ν
Re_{eff}	effective Reynolds number
U	free-stream velocity ($\mathrm{m\,s^{-1}}$)
u, v	velocity components ($\mathrm{m\,s^{-1}}$)
$\mathbf{V}$	velocity vector
$\mathbf{V}^*$	intermediate velocity
$\delta\bar{V}$	material parcel volume ($\mathrm{m^3}$)
Greek	
Δt	time step (s)
$\Delta x, \Delta y$	grid spacings (m)
ν	kinematic viscosity ($\mathrm{m^2\,s^{-1}}$)
ν_{num}	numerical viscosity
ρ	fluid density ($\mathrm{kg\,m^{-3}}$)
σ	time-step safety factor
ω_z	spanwise vorticity ($\mathrm{s^{-1}}$)

5. Computational Setup

The domain is a rectangle $L_x \times L_y = 70\,\mathrm{m} \times 35\,\mathrm{m}$, discretised on a uniform Cartesian mesh of 220×110 points, corresponding to the 24,200-point mesh used in the validation plot. The cylinder of radius $R = 2\,\mathrm{m}$ is centred at $(L_x/4,\ L_y/2 + 0.1\Delta y)$; the small vertical offset breaks the initial symmetry and assists the development of asymmetric shedding where the flow is physically unsteady. The downstream fetch of approximately 52.5 m, or just over 13 diameters, allows shed vortical structures to convect away from the body before reaching the outlet boundary.

Fluid properties are fixed at $\rho = 1.105\,\mathrm{kg\,m^{-3}}$ and $\nu = 0.03\,\mathrm{m^2\,s^{-1}}$. For each case the free-stream velocity is $U = Re\,\nu/D$. Reynolds numbers surveyed for the drag validation span $Re \in \{10^3,\ 5\times10^3,\ 10^4,\ 5\times10^4,\ 10^5,\ 2\times10^5,\ 4\times10^5,\ 6\times10^5,\ 10^6\}$. Additional low- and moderate-Reynolds-number visualisations are included to assess whether the computed wake structure is qualitatively consistent with expected cylinder-flow behaviour.

6. Results and Discussion

6.1. Drag Coefficient: Comparison with Experiment

Figure 2 compares the predicted C_D against the experimental correlation of Schlichting [1] over several decades of Reynolds number. The numerical data are plotted as the present CFD result on the 24,200-point Cartesian mesh. The error bars shown on the numerical curve represent the oscillatory range of the instantaneous drag coefficient where periodic shedding is present, rather than measurement uncertainty.

The solver captures the viscous-dominated low-

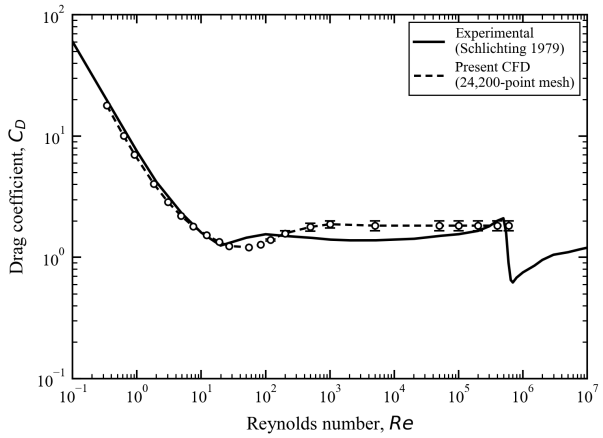


Figure 2: Drag coefficient C_D vs. Reynolds number Re. Solid line: experimental correlation [1]. Dashed line: present solver on the 24,200-point mesh. Error bars indicate the oscillatory range of instantaneous pressure drag in cases where periodic vortex shedding is present.

Reynolds-number trend, including the strong reduction of C_D with increasing Re and the local minimum near $C_D \approx 1.2$–1.4 at $Re \approx 20$–50. Beyond $Re \approx 100$, the numerical solution progressively over-predicts C_D relative to the experimental curve. This is consistent with the two-dimensional formulation: at moderate and high Re, real cylinder wakes develop spanwise instabilities and secondary vortex interactions that modify the base pressure and reduce the mean drag. Since the present solver cannot represent spanwise variation, the wake remains artificially coherent and the rear-face pressure deficit is systematically over-estimated.

The drag crisis near $Re \approx 9 \times 10^5$, in which C_D drops sharply due to boundary-layer transition and delayed separation, is absent from the present numerical result. The uniform Cartesian mesh lacks the near-wall refinement required to resolve the thin transitional boundary layer, and no transition or turbulence model is included. Although the solver advances the unsteady Navier–Stokes equations without a turbulence closure, the grid is not fine enough for the calculation to be interpreted as a resolved high-Reynolds-number DNS.

6.2. Vorticity Field and Wake Development

The vorticity field is used as a qualitative check on the computed wake physics. The out-of-plane vorticity is

$$\omega_z = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}. \tag{18}$$

With the standard right-handed sign convention, positive ω_z corresponds to anticlockwise rotation and negative ω_z corresponds to clockwise rotation.

Figure 3 shows the low-Reynolds-number wake at $Re = 26$. At this Reynolds number the separated shear layers remain organised into a steady, approximately symmetric wake rather than a fully periodic vortex street. The upper and lower shear layers have opposite signs of vorticity, as expected for flow separating from opposite sides of a circular bluff body.

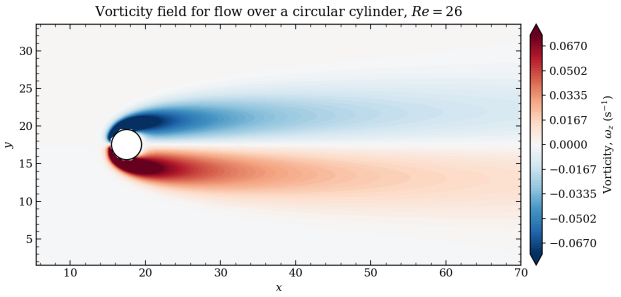


Figure 3: Computed vorticity field at $Re = 26$. The opposite-signed upper and lower shear layers show the expected separated wake structure behind the circular cylinder. At this Reynolds number the flow is still represented as a steady separated wake rather than a fully periodic vortex street.

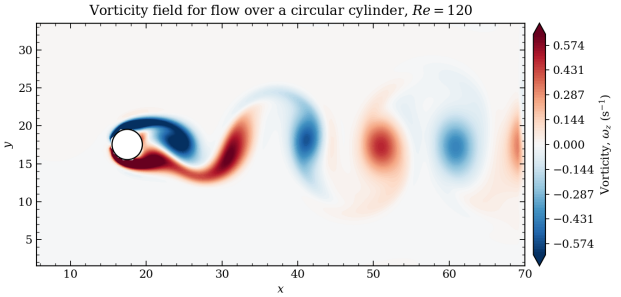


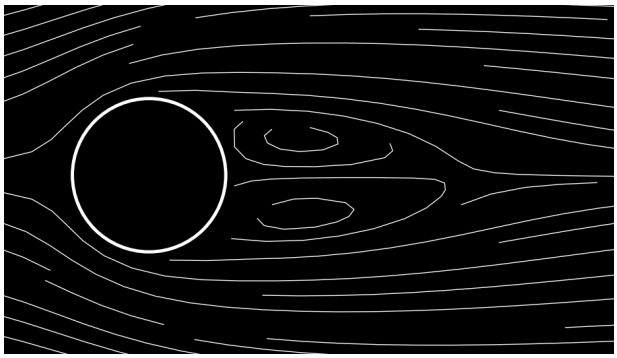
Figure 4: Computed vorticity field at $Re = 120$. Alternating positive and negative vorticity regions are shed behind the cylinder, indicating the periodic release of anticlockwise and clockwise vortices from the separated shear layers.

Figure 4 shows the corresponding vorticity field at $Re = 120$. Here the wake is no longer a steady symmetric structure. Alternating regions of positive and negative vorticity are shed downstream, which is consistent with the expected development of a Kármán-type wake. The visual pattern is important because it confirms that the solver is not only producing drag values, but is also producing the correct qualitative mechanism: clockwise and anticlockwise vortices are shed alternately from the bluff body.

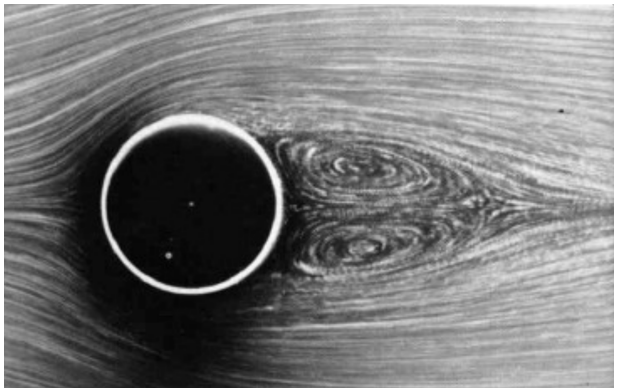
These vorticity plots therefore provide a visual consistency check on the flow regime. The $Re = 26$ case demonstrates a steady separated wake, while the $Re = 120$ case demonstrates the expected alternating vortex shedding. This transition in the computed wake structure is consistent with the known physical progression of cylinder flow as inertial effects increase.

6.3. Streamline Visualisation

The streamline plots in Figure 5 are included as a second qualitative visualisation of the computed wake. The black-background rendering is not used as a quantitative validation metric; rather, it is intended to make the recirculation structure visually comparable with classical smoke-line or streakline visualisations of flow past a cylinder.



(a) Computed streamline field.



(b) Reference flow visualisation from Van Dyke's *An Album of Fluid Motion* [4].

Figure 5: Qualitative streamline comparison for flow past a circular cylinder. The computed black-background plot shows the closed recirculation region downstream of the cylinder, while the reference visualisation illustrates the same physical wake feature in a laboratory-style flow image.

In the computed streamline field, white streamlines curve around the black cylinder and enclose a recirculating region immediately behind the body. This is the streamline signature of separated flow behind a bluff body. The reference visualisation shows a similar closed wake region, providing an intuitive check that the numerical solution is producing a physically recognisable separated wake topology.

6.4. Convergence Behaviour

At sufficiently low Reynolds number the flow tends towards a steady separated state within the iteration limit. At higher Reynolds number, the drag coefficient exhibits oscillations once periodic vortex shedding develops. For this reason, the error bars on the drag validation plot are interpreted as the oscillatory range of instantaneous pressure drag rather than as statistical uncertainty. The current solver therefore provides both scalar comparison through C_D and field-level comparison through vorticity and streamline visualisation.

7. Conclusions

A two-dimensional incompressible Navier–Stokes solver has been developed, implemented, and validated against the experimental drag coefficient correlation for flow past a circular cylinder. The cylinder is represented on a fixed Cartesian grid through a mask-based immersed-boundary treatment, avoiding the need for body-fitted meshing at the cost of an approximate staircase representation of the curved surface. Pressure–velocity coupling is handled using Chorin's projection method, and convective derivatives are evaluated using a second-order directional upwind-biased stencil.

The solver reproduces the broad low-Reynolds-number drag trend and produces the expected qualitative wake development. At $Re = 26$, the vorticity field shows a steady separated wake with opposite-signed shear layers. At $Re = 120$, alternating regions of positive and negative vorticity are shed from the cylinder, indicating the formation of a Kármán-type wake. The streamline visualisations also show a closed recirculation region behind the bluff body, consistent with classical flow visualisations.

At higher Reynolds numbers the two-dimensional formulation is the primary source of drag over-prediction, as the spanwise wake instabilities that reduce drag in real three-dimensional flow cannot develop in a planar simulation. The drag crisis near $Re \approx 9 \times 10^5$ is not captured because the uniform Cartesian grid does not resolve the thin transitional boundary layer and no transition model is present. Taken together, the results constitute a physically consistent code-development baseline. Future work will introduce active flow control via surface jet blowing, characterising drag reduction and wake modification as functions of jet position and jet momentum coefficient.

References

[1] H. Schlichting, *Boundary Layer Theory*, 7th ed. McGraw-Hill, New York, 1979.

[2] A. J. Chorin, "Numerical solution of the Navier–Stokes equations," *Math. Comput.*, vol. 22, no. 104, pp. 745–762, 1968.

[3] J. D. Anderson, *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill, New York, 1995.

[4] M. Van Dyke, *An Album of Fluid Motion*. The Parabolic Press, Stanford, 1982.

A. Full Solver Source Code

The complete Python solver is included below for reproducibility. The file `flow_past_cylinder.py` contains the exact solver used to generate the drag and vorticity results.

```python
import numpy as np
import matplotlib.pyplot as py
from matplotlib.patches import Circle
import os

## - - FLOW PAST A CIRCULAR CYLINDER - - #

# First order in time explicit solver using Chorin's Projection

# Second order in space

# ------------------------------------------ Domain length = 60
    ------------------------------------------------------------
Lx =  70 # meters
Ly = 35 # meters
nx = 200
ny = 100
dx = Lx/(nx-1)
dy = Ly/(ny-1)
nt = 4200
dt = 0.0001
rho = 1.105
nu = 0.03
#U = 1

# ---- Circle Dimensions ----
Circle_radius = 2
Diameter = 2*Circle_radius

SAVEEVERY = 50
VARIABLESTARTING = 1300
MAXITER = 50000
ConvergingLimit = 50

# -- REYNOLDS NUMBER LIST --
Re_list = [48,58,69,85,96,120,130,140,155]


# ---- Circle positions ----
x_centre = Lx/4
y_centre = Ly/2 +0.1*dy
i_centre = int((x_centre)/dx) # positioned at x = Lx/2
j_centre = int((y_centre)/dy) # positioned at y = Ly/2

# --- circle surface ----

n_circle = 80
theta = np.linspace(0,2*np.pi,n_circle, endpoint=False)
dtheta = (2*np.pi)/n_circle

# --- CREATING DOMAIN ----
x = np.linspace(0, Lx, nx)
y = np.linspace(0, Ly, ny)

X, Y = np.meshgrid(x, y)


# -- Creating Circle ----
circlepatch = (X-x_centre)**2 + (Y-y_centre)**2 <= Circle_radius**2

# -- C A S E --
for Re_case in Re_list:

    U = (Re_case*nu)/Diameter
    Re = (U*(Diameter))/nu

    #- SAVE SETUP -
    casefolder = f"Re_finer={Re}"
    os.makedirs(casefolder, exist_ok=True)

    cd_folder = f"{casefolder}/Drag_Frames"
    Pressure_folder = f"{casefolder}/Pressure_Frames"
    vorticity_folder = f"{casefolder}/Vorticity_Frames"

    os.makedirs(cd_folder, exist_ok=True)
    os.makedirs(Pressure_folder, exist_ok=True)
    os.makedirs(vorticity_folder, exist_ok=True)

    # INITIALISING VELOCITY AND PRESSURE
```

```
    u = np.zeros((ny, nx))
    v = np.zeros((ny, nx))
    p = np.zeros((ny, nx))

    # initialise whole domain with moving flow
    u = np.zeros((ny, nx)) + U




    # ------- VELOCITY BOUNDARY CONDITIONS ---------

    # left inlet
    u[:, 0] = U
    v[:, 0] = 0

    # ghost inlet
    u[:, 1] = U
    v[:, 1] = 0

    # right outlet
    u[:, -1] = u[:, -3]
    v[:, -1] = v[:, -3]

    #ghost outlet
    u[:, -2] = u[:, -3]
    v[:, -2] = v[:, -3]

    # bottom Neumann/open boundary
    u[0, :] = u[2, :]
    v[0, :] = v[2, :]

    # ghost bottom
    u[1, :] = u[2, :]
    v[1, :] = v[2, :]


    # top Neumann/open boundary
    u[-1, :] = u[-3, :]
    v[-1, :] = v[-3, :]

    # ghost top
    u[-2, :] = u[-3, :]
    v[-2, :] = v[-3, :]

    # PRESSURE BC ------------- DIRECHLECT UPPER,LOWER,INLET --------- NEUMANN OUTLET
    p[:, 0] = p[:, 2] #left
    p[:, 1] = p[:, 2] #ghost left

    p[:, -1] = 0 #outlet
    p[:, -2] = 0 #ghost outlet

    p[0, :] = p[2, :] # bottom
    p[1, :] = p[2, :] # ghost bottom

    p[-1, :] = p[-3, :] #top
    p[-2, :] = p[-3, :] # ghost top


    # --- No SLIP CONDITION FOR CIRCLE ---

    u[circlepatch] = 0
    v[circlepatch] = 0



    #Calculated quantities

    vorticity = np.zeros_like(u)

    vorticity[1:-1,1:-1] = (v[1:-1,2:] - v[1:-1,0:-2])/(2*dx)  -  (u[2:,1:-1] - u[0:-2,1:-1])/(2*dy)

    dynamic_pressure = 0.5 * rho * U**2

    P_ref = np.mean(p[:,2])



    #- AERODYNAMIC FORCES -
    Drag_Coeff = []
    iterations = []
    Surface_Cp = []

```

```
163
164     # -- PRESSURE AGAINST THETA GRAPH SET UP --
165     cpfig2, (ax2) = py.subplots(1,1)
166     Cpplot, = ax2.plot([],[], 'ko-', lw = 0.5,markersize = 2)
167
168     ax2.set_title(f"Pressure Coefficient of a circular cylinder at Re={Re:.0f}")
169     ax2.set_xlabel(r'$\theta$ (rad)')
170     ax2.set_ylabel(r'$C_P$')
171     ax2.set_xlim(min(theta),max(theta))
172     ax2.tick_params(direction='in')
173     # -- AERODYNAMIC FORCES GRAPH SET UP --
174
175     forcefig, (forceaxis) = py.subplots(1,1)
176     forceaxis.set_title(r'Variation of $C_D$ with iterations.')
177     dragplot, = forceaxis.plot([],[], 'k-', lw = 0.5)
178
179     forceaxis.set_xlabel('iterations')
180     forceaxis.set_ylabel(r'$C_D$')
181     forceaxis.tick_params(direction='in')
182     forceaxis.grid()
183
184     # CONTOUR PLOT FOR VORTICITY
185
186     contourfig1, (ax1) = py.subplots(1, 1)
187     vorticityplot = ax1.pcolormesh(X, Y, vorticity, cmap='seismic')
188     ax1.set_title(f"Flow over a circular cylinder, Re={Re:.3f}.")
189     cbar = contourfig1.colorbar( vorticityplot, ax=ax1, label='Vorticity (1/s)', shrink = 0.6, fraction =
            0.08)
190     ax1.set_xlim(0, Lx)
191     ax1.set_ylim(0, Ly)
192     ax1.set_xlabel('x')
193     ax1.set_ylabel('y')
194     ax1.set_aspect('equal')
195     ax1.set_yticks(np.arange(0,Ly,4))
196     ticks1 = cbar.get_ticks()
197     cbar.set_ticks(ticks1[::3])
198     ax1.tick_params(direction='in')
199
200
201
202     # CIRCLE PATCH
203
204     circle = Circle((x_centre,y_centre), Circle_radius, facecolor = 'white', edgecolor = 'white' , zorder =
            11)
205
206     ax1.add_patch(circle)
207
208
209     def zeropressuregrad(nx,ny,circlepatch,p,X,Y,):
210
211
212         for j in range(1,ny-1):
213             for i in range(1,nx-1):
214                 if circlepatch[j,i]:
215                     xdistance = X[j,i] - x_centre
216                     ydistance = Y[j,i] - y_centre
217                     if abs(xdistance) > abs(ydistance):
218                         if xdistance > 0:
219                             p[j,i] = p[j,i+1]
220                         else:
221                             p[j,i] = p[j,i-1]
222                     else:
223                         if ydistance > 0:
224                             p[j,i] = p[j+1,i]
225                         else:
226                             p[j,i] = p[j-1,i]
227         return p
228
229
230     def buildup(rho, dt, u, v, dx, dy):
231         global ny,nx
232         b = np.zeros((ny, nx))
233         b[2:-2, 2:-2] = (rho / dt) * ((u[2:-2, 3:-1] - u[2:-2, 1:-3]) / (2 * dx)+(v[3:-1, 2:-2] - v[1:-3,
                2:-2]) / (2 * dy))
234         return b
235
236
237     def poissonpressure(p, b, dx, dy, l1normtarget, MAXITER):
238
239         l1norm = 1
240         iteration = 0
241         while l1norm > l1normtarget and iteration < MAXITER:
242
243             pn = p.copy()
```

```
            p[2:-2, 2:-2] = (
            dy**2 * (pn[2:-2, 3:-1] + pn[2:-2, 1:-3])
            + dx**2 * (pn[3:-1, 2:-2] + pn[1:-3, 2:-2])
            - b[2:-2, 2:-2] * dx**2 * dy**2
            ) / (2 * (dx**2 + dy**2))

            # PRESSURE BC ------------- DIRECHLECT UPPER,LOWER,INLET --------- NEUMANN OUTLET
            p[:, 0] = p[:, 2] #left
            p[:, 1] = p[:, 2] #ghost left

            p[:, -1] = 0 #outlet
            p[:, -2] = 0 #ghost outlet

            p[0, :] = p[2, :] # bottom
            p[1, :] = p[2, :] # ghost bottom

            p[-1, :] = p[-3, :] #top
            p[-2, :] = p[-3, :] # ghost top

            # --- ZERO PRESSURE GRAD ON CIRCLE WALL ---
            zeropressuregrad(nx,ny,circlepatch,p,X,Y,)

            l1norm = np.linalg.norm(p - pn) / (np.linalg.norm(pn) + 1e-12)
            iteration += 1

        return p,iteration,l1norm


    def flowovercylinder(u,v,nu,rho,p,dt,dx,dy,nt,U):

        global velocityvectors, circlepatch, VARIABLESTARTING, ConvergingLimit
        CONVERGING_COUNTER = 0


        b = np.zeros((ny, nx))

        for n in range(nt):
            SAFETY_FACTOR = 0.3
            dt_hyperbolic = (SAFETY_FACTOR * 1) / (
            (np.max(np.abs(u)) / dx) + (np.max(np.abs(v)) / dy)
            )
            dt_parabolic = (SAFETY_FACTOR * 0.5) / (
            nu * ((1 / dx**2) + (1 / dy**2))
            )
            dt = min(dt_hyperbolic, dt_parabolic)

            un = u.copy()
            vn = v.copy()

            # Second-order directional upwind derivatives
            # Backward stencil: (3f_i - 4f_{i-1} + f_{i-2}) / (2dx)
            # Forward stencil:  (-3f_i + 4f_{i+1} - f_{i+2}) / (2dx)

            du_dx_backward = (
            3 * un[2:-2, 2:-2]
            - 4 * un[2:-2, 1:-3]
            + un[2:-2, 0:-4]
            ) / (2 * dx)

            du_dx_forward = (
            -3 * un[2:-2, 2:-2]
            + 4 * un[2:-2, 3:-1]
            - un[2:-2, 4:]
            ) / (2 * dx)

            du_dy_backward = (
            3 * un[2:-2, 2:-2]
            - 4 * un[1:-3, 2:-2]
            + un[0:-4, 2:-2]
            ) / (2 * dy)

            du_dy_forward = (
            -3 * un[2:-2, 2:-2]
            + 4 * un[3:-1, 2:-2]
            - un[4:, 2:-2]
            ) / (2 * dy)

            dv_dx_backward = (
            3 * vn[2:-2, 2:-2]
            - 4 * vn[2:-2, 1:-3]
            + vn[2:-2, 0:-4]
            ) / (2 * dx)

```

```
            dv_dx_forward = (
            -3 * vn[2:-2, 2:-2]
            + 4 * vn[2:-2, 3:-1]
            - vn[2:-2, 4:]
            ) / (2 * dx)

            dv_dy_backward = (
            3 * vn[2:-2, 2:-2]
            - 4 * vn[1:-3, 2:-2]
            + vn[0:-4, 2:-2]
            ) / (2 * dy)

            dv_dy_forward = (
            -3 * vn[2:-2, 2:-2]
            + 4 * vn[3:-1, 2:-2]
            - vn[4:, 2:-2]
            ) / (2 * dy)

            du_dx = np.where(un[2:-2, 2:-2] < 0, du_dx_forward, du_dx_backward)
            du_dy = np.where(vn[2:-2, 2:-2] < 0, du_dy_forward, du_dy_backward)

            dv_dx = np.where(un[2:-2, 2:-2] < 0, dv_dx_forward, dv_dx_backward)
            dv_dy = np.where(vn[2:-2, 2:-2] < 0, dv_dy_forward, dv_dy_backward)

            u[2:-2, 2:-2] = (
            un[2:-2, 2:-2]
            - un[2:-2, 2:-2] * dt * du_dx
            - vn[2:-2, 2:-2] * dt * du_dy
            + nu
            * (
            dt / dx**2
            * (un[2:-2, 3:-1] - 2 * un[2:-2, 2:-2] + un[2:-2, 1:-3])
            + dt / dy**2
            * (un[3:-1, 2:-2] - 2 * un[2:-2, 2:-2] + un[1:-3, 2:-2])
            )
            )

            v[2:-2, 2:-2] = (
            vn[2:-2, 2:-2]
            - un[2:-2, 2:-2] * dt * dv_dx
            - vn[2:-2, 2:-2] * dt * dv_dy
            + nu
            * (
            dt / dx**2
            * (vn[2:-2, 3:-1] - 2 * vn[2:-2, 2:-2] + vn[2:-2, 1:-3])
            + dt / dy**2
            * (vn[3:-1, 2:-2] - 2 * vn[2:-2, 2:-2] + vn[1:-3, 2:-2])
            )
            )

            # Enforce BC at each time step

            # ------------ VELOCITY BOUNDARY CONDITIONS --------------

            # left inlet
            u[:, 0] = U
            v[:, 0] = 0

            # ghost inlet
            u[:, 1] = U
            v[:, 1] = 0

            # right outlet
            u[:, -1] = u[:, -3]
            v[:, -1] = v[:, -3]

            #ghost outlet
            u[:, -2] = u[:, -3]
            v[:, -2] = v[:, -3]

            # bottom Neumann/open boundary
            u[0, :] = u[2, :]
            v[0, :] = v[2, :]

            # ghost bottom
            u[1, :] = u[2, :]
            v[1, :] = v[2, :]


            # top Neumann/open boundary
            u[-1, :] = u[-3, :]
            v[-1, :] = v[-3, :]

            # ghost top
```

```
            u[-2, :] = u[-3, :]
            v[-2, :] = v[-3, :]

            # ------------- BOUNDARY CONDITIONS FOR SQUARE CYLINDER --------------

            u[circlepatch] = 0
            v[circlepatch] = 0


            # --------------- PROJECTION STEP -- COMPUTING PRESSURE USING DIVERGENCE FREE CONSTRAINT
                 ------------------------


            b = buildup(rho, dt, u, v, dx, dy)
            p,iteration,l1norm = poissonpressure(p, b, dx, dy, 1e-5, MAXITER)

            if iteration ==1:
                CONVERGING_COUNTER += 1
            else:
                CONVERGING_COUNTER = 0

            if CONVERGING_COUNTER >= ConvergingLimit:
                print("Reached steady state!")
                break


            u[2:-2,2:-2] = u[2:-2,2:-2] - (dt/(2*rho*dx))*(p[2:-2,3:-1] - p[2:-2,1:-3])

            v[2:-2,2:-2] = v[2:-2,2:-2] - (dt/(2*rho*dy))*(p[3:-1,2:-2] - p[1:-3,2:-2])

            #Enforce Neumann and Direchlet BC again...

            # ------------ VELOCITY BOUNDARY CONDITIONS --------------

            # left inlet
            u[:, 0] = U
            v[:, 0] = 0

            # ghost inlet
            u[:, 1] = U
            v[:, 1] = 0

            # right outlet
            u[:, -1] = u[:, -3]
            v[:, -1] = v[:, -3]

            #ghost outlet
            u[:, -2] = u[:, -3]
            v[:, -2] = v[:, -3]

            # bottom Neumann/open boundary
            u[0, :] = u[2, :]
            v[0, :] = v[2, :]

            # ghost bottom
            u[1, :] = u[2, :]
            v[1, :] = v[2, :]



            # top Neumann/open boundary
            u[-1, :] = u[-3, :]
            v[-1, :] = v[-3, :]

            # ghost top
            u[-2, :] = u[-3, :]
            v[-2, :] = v[-3, :]

            # ------------- BOUNDARY CONDITIONS FOR SQUARE CYLINDER --------------

            u[circlepatch] = 0
            v[circlepatch] = 0

            #Calculated quantities

            #- VORTICITY -
            vorticity[1:-1,1:-1] = (v[1:-1,2:] - v[1:-1,0:-2])/(2*dx)  -  (u[2:,1:-1] - u[0:-2,1:-1])/(2*dy)

            #- AERODYNAMIC FORCES -

            if n > 1:
                vorticity_MAX_plot = max(np.max(vorticity),1e-12)

                vorticityplot.set_array(vorticity.ravel())
```

```
                vorticityplot.set_clim(-vorticity_MAX_plot,vorticity_MAX_plot)

                cbar.set_ticks(np.linspace(-vorticity_MAX_plot, vorticity_MAX_plot, 6))
                contourfig1.savefig(
                f"{vorticity_folder}/vorticity_frame_{n:05d}.png",
                dpi=150
                )
                CFL = np.max(np.abs(u))*dt/dx + np.max(np.abs(v))*dt/dy
                print("CFL =", CFL,"umax =", np.max(np.abs(u)),"vmax =", np.max(np.abs(v)),"pmax =",
                     np.max(np.abs(p)), "pressure iteratiosn =",iteration, "l1norm=",l1norm)

            if n>VARIABLESTARTING and n % SAVEEVERY ==0:

                    Surface_Cp = []
                    Drag_Force = 0
                    Lift_Force = 0

                    dynamic_pressure = 0.5 * rho * U**2

                    P_ref = np.mean(p[:,2])
                    for delta in theta:

                        # -- SURFACE PRESSURE --
                        xi = i_centre + int(((Circle_radius+dx)*np.cos(delta))/dx)
                        yi = j_centre + int(((Circle_radius+dy)*np.sin(delta))/dy)
                        pressure_probe_value = p[yi,xi]
                        Cp = (pressure_probe_value - P_ref)/dynamic_pressure
                        Surface_Cp.append(Cp)

                        # -- DRAG --
                        Drag_Force = Drag_Force -Circle_radius* pressure_probe_value*dtheta*np.cos(delta)
                        CD = (Drag_Force)/(dynamic_pressure*Diameter)

                    Drag_Coeff.append(CD)
                    iterations.append(n)

                    # --- UPDATING PLOTS -----


                    Cpplot.set_data(theta,Surface_Cp)
                    ax2.set_ylim(min(Surface_Cp)-0.3, max(Surface_Cp)+1)

                    dragplot.set_data(iterations,Drag_Coeff)
                    forceaxis.set_ylim(min(Drag_Coeff)-0.5, max(Drag_Coeff)+0.5)
                    forceaxis.set_xlim(VARIABLESTARTING,max(iterations)+1)


                    cpfig2.savefig(
                        f"{Pressure_folder}/pressurecoeff_frame_{n:05d}.png",
                        dpi=100
                    )

                    forcefig.savefig(
                        f"{cd_folder}/drag_frame_{n:05d}.png",
                        dpi=100
                    )


        return u, v, p, n


    u, v, p, iterationnumber = flowovercylinder(u,v,nu,rho,p,dt,dx,dy,nt,U)




py.show()
```

Listing 4: Complete Python solver: flow past a circular cylinder using Chorin's projection method, a mask-based immersed-boundary treatment, second-order directional upwind convection, adaptive time stepping, vorticity output, and pressure-drag evaluation.