AI-Assisted Coding Project in Python Python-Based URANS Simulation of Transitional Flow over NACA Airfoils Using the γ–Re_θ–SST Model at Moderate Reynolds Number

Vigneshwara Koka Balaji
CMS@TUBAF matr.no. 71624

July 11, 2025

Abstract

This project investigates the use of AI-assisted programming for developing a 2D unsteady incompressible Navier- Stokes solver capable of modeling laminar-to-turbulent transition over NACA airfoils. Using a structured two-prompt approach, OpenAIs ChatGPT-4o (mini-high variant) was employed to generate a fully modular Python-based solver implementing the URANS γ–Re_θ–SST transition model. The solver includes structured mesh generation, Crank-Nicolson time integration, sparse matrix-based FVM discretization, and automated postprocessing for aerodynamic analysis and visualization, enables robust unsteady simulation of separation onset, transition propagation, and wake evolution. Simulations were conducted for multiple NACA geometries (e.g., NACA 4412, NACA 0018) to validate the solver across different airfoil classes. Results include pressure and velocity field contours, animated streamlines, and identification of separation bubbles. The study demonstrates the potential of AI-assisted scientific computing workflows in accelerating the development of advanced transitional CFD solvers across a family of aerodynamic shapes.

Keywords: URANS, γ–Re_θ transition model, k–ω SST Model, NACA Airfoils, Python solver, AI-assisted programming, ChatGPT, Transitional flow analysis, Computational Fluid Dynamics (CFD).

1 Introduction

This report presents the development and analysis of a modular Python-based CFD solver for simulating two-dimensional incompressible transitional flow over NACA- series airfoils at moderate Reynolds numbers. Transitional aerodynamic regimes where boundary layers evolve from laminar to turbulent are notoriously sensitive to surface curvature, pressure gradients, and freestream conditions. Capturing such behavior is essential for accurately predicting aerodynamic performance parameters such as lift, drag, and stall onset.

To address these challenges, the solver employs the Unsteady Reynolds-Averaged Navier Stokes (URANS) formulation, augmented by the γ-Re_θ transition model and the SST k-ω turbulence closure. Discretized using the finite volume method on a structured multi-zone mesh, the solver incorporates a fully implicit Crank Nicolson time integration scheme for improved numerical stability and second-order accuracy. The solver was developed using AI-assisted programming via structured prompts to ChatGPT, which produced a complete and executable code base including pre- and postprocessing modules.

The following subsections introduce the physical background of transitional flows, relevance of NACA airfoils in this regime, governing fluid equations, turbulence-transition modeling strategies, and the numerical framework used in the solver.

1.1 Transitional Aerodynamics and Relevance of NACA Airfoils

In aerodynamic flows over lifting surfaces such as airfoils, the boundary layer often experiences transition from a laminar to a turbulent state. This transition process is critical, as it directly influences drag characteristics, lift generation, and flow separation behavior. Transitional flow regimes are particularly dominant at moderate Reynolds numbers ($Re \approx 10^5$-10^6), which are common in applications ranging from unmanned aerial vehicles (UAVs) to wind turbine blades and light aircraft. In this regime, even small

variations in surface curvature or pressure gradient can significantly shift the transition onset location and trigger complex unsteady phenomena such as laminar separation bubbles or vortex shedding.

NACA airfoils - especially the symmetric 0012 and 0018, and cambered variants such as 2412 or 4412 - are widely used in aerodynamic studies due to their smooth analytical geometry and the availability of extensive experimental and numerical data. These profiles are ideal candidates for validating transition-sensitive turbulence models. For instance, cambered airfoils like NACA 4412 exhibit pressure-induced transition over the suction surface, whereas symmetric profiles such as NACA 0018 often demonstrate separation-induced transition at higher angles of attack. Because their shapes are analytically defined, mesh generation is highly reproducible and adaptable to structured grid strategies.

Thus, NACA profiles provide an excellent framework to study transitional flow physics and evaluate turbulence-transition models such as the γ-Re_θ formulation, especially when deployed in open-source, Python-based CFD solvers like the one developed in this project.

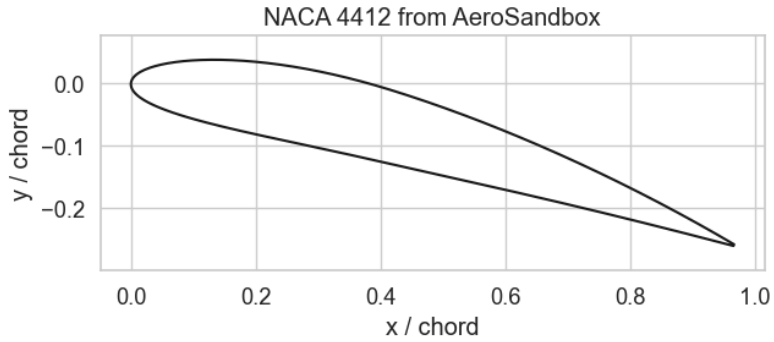

Figure 1: Airfoil shape of NACA 4412 generated using AeroSandbox.

1.2 Challenges in Transition Modeling

Capturing the physics of laminar-to-turbulent transition remains one of the most intricate challenges in computational fluid dynamics. Conventional turbulence models such as k-ε or SST k-ω often assume fully turbulent flow from the leading edge, which limits their ability to predict natural transition or separation-induced transition. As a result, these models tend to overpredict skin friction drag and fail to capture transitional structures like laminar separation bubbles.

Transition is governed by a cascade of mechanisms-disturbance amplification, nonlinear growth, breakdown, and reattachment-all of which depend on both local and non-local flow conditions. Factors like freestream turbulence intensity, surface curvature, adverse pressure gradients, and surface roughness critically influence when and where transition occurs. Resolving these mechanisms directly requires expensive high-fidelity methods such as Direct Numerical Simulation (DNS) or wall-resolved Large Eddy Simulation (LES), which are computationally prohibitive for practical engineering problems.

To address this, dedicated transition models such as the Langtry-Menter γ-Re_θ framework have been developed. These models introduce two additional scalar transport equations: one for intermittency (γ) and one for the transition momentum-thickness Reynolds number (Re_θ). Together, they enable the turbulence model to activate gradually in regions where transition is physically expected. This approach provides a computationally affordable yet physically grounded method to simulate transition in a time-accurate URANS framework, especially in cases involving separation and reattachment.

1.3 Unsteady Reynolds Averaged Navier Stokes (URANS) with γ -Re_θ -SST Framework

The Unsteady Reynolds-Averaged Navier-Stokes (URANS) equations are the time dependent extended version of RANS equations, while traditional RANS equations solve for steady state flows. URANS models still use time-averaged equations but include the time derivative of the mean flow variables. This makes them especially suitable for moderately unsteady flows where time-resolved but not fully resolved turbulence is a need. Compared to other turbulent models such as DNS or LES, URANS provides a equilibrium between physical fidelity and computational effort.

The γ -Re_θ model introduces two additional transport equations: one for intermittency γ, which governs the local onset and progression of turbulence, and one for the thickness Reynolds number Re_θ, which encodes the empirical relationship between pressure gradients and transition location. This framework

has proven effective in capturing natural and separation-induced transition in a range of aerodynamic configurations.

The SST k-ω turbulence model is employed post-transition due to its ability to accurately predict wall-bounded shear flows and free shear layers. Together, the combined framework provides better physically accurate simulations of transitional airfoil flows while still remaining suitable for time-accurate and computationally efficient simulations.

1.4 Governing Equations for Incompressible Transitional Flow

The simulation framework solves the unsteady, incompressible Navier-Stokes equations in two dimensions, extended to incorporate turbulence and transition phenomena using a hybrid URANS approach. The incompressibility condition is enforced via the continuity equation:

$$\nabla \cdot \vec{u} = 0 \tag{1}$$

*Source: [**chen2022**]*

The momentum transport equations for the velocity vector field $\vec{u} = (u, v)$ take the form:

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = -\frac{1}{\rho}\nabla p + \nabla \cdot [(\nu + \nu_t)\nabla \vec{u}] \tag{2}$$

*Source: [**chen2022**]* Here, ρ is the fluid density, ν is the kinematic molecular viscosity, and ν_t is the turbulent eddy viscosity derived from the SST k-ω model and transition intermittency γ via:

$$\nu_t = \frac{\gamma k}{\omega} \tag{3}$$

*Source: [**langtry2009**]* To capture transitional behavior, the solver augments the momentum equations with scalar transport equations for:

- **Turbulent kinetic energy (k)**:

$$\frac{\partial k}{\partial t} + \vec{u} \cdot \nabla k = P_k - \beta^* \omega k + \nabla \cdot [(\nu + \sigma_k \nu_t)\nabla k] \tag{4}$$

  *Source: [**SSTequations**]*

- **Specific dissipation rate (ω)**:

$$\frac{\partial \omega}{\partial t} + \vec{u} \cdot \nabla \omega = \alpha \frac{\omega}{k} P_k - \beta \omega^2 + \nabla \cdot [(\nu + \sigma_\omega \nu_t)\nabla \omega] \tag{5}$$

  *Source: [**SSTequations**]*

- **Transition momentum thickness Reynolds number (Re_θ)**:

$$\frac{\partial Re_\theta}{\partial t} + \vec{u} \cdot \nabla Re_\theta = \nabla \cdot \left(\frac{\nu}{Pr_\theta} \nabla Re_\theta \right) + P_{Re_\theta} \tag{6}$$

  *Source: [**langtry2006a**]*

- **Intermittency (γ)**:

$$\frac{\partial \gamma}{\partial t} + \vec{u} \cdot \nabla \gamma = \nabla \cdot \left(\frac{\nu}{Pr_\gamma} \nabla \gamma \right) + P_\gamma - D_\gamma \tag{7}$$

  *Source: [**langtry2006a**]*

Together, these equations form a closed system capable of resolving boundary layer evolution from laminar inflow through transitional breakdown to fully turbulent reattachment. The inclusion of the γ-Re_θ model ensures sensitivity to local flow instabilities while maintaining robustness within the URANS framework.

1.5 Numerical Formulation

The solver employs a fully implicit finite volume discretization for all governing equations on a structured mesh. Time integration is performed using the Crank-Nicolson θ-scheme, which provides second-order accuracy and improved numerical stability. For all scalar transport equations and momentum equations, a generic transport formulation is discretized as follows:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\phi \vec{u}) = \nabla \cdot (\Gamma_\phi \nabla \phi) + S_\phi \tag{8}$$

*Source: [***langtry2006a***]* Using the Crank-Nicolson method with $\theta = 0.5$, the semi-discrete form becomes:

$$\left(\frac{\phi^{n+1} - \phi^n}{\Delta t}\right) + \theta \nabla \cdot (\phi^{n+1} \vec{u}^{n+1}) + (1-\theta)\nabla \cdot (\phi^n \vec{u}^n) = \theta \nabla \cdot (\Gamma_\phi \nabla \phi^{n+1}) + (1-\theta)\nabla \cdot (\Gamma_\phi \nabla \phi^n) + \bar{S}_\phi \tag{9}$$

This formulation is applied to all scalar fields, including k, ω, γ, and Re_θ, ensuring strong temporal coupling between turbulence and transition fields. The discretization is implemented via structured finite volume stencils and stored in sparse matrix format.

1.5.0.1 Pressure-Velocity Coupling: The solver uses a monolithic pressure-correction method. First, the momentum equations are discretized and solved to obtain provisional velocities $\vec{u}^*$ that do not yet satisfy continuity. A pressure Poisson equation is then derived from the discrete continuity condition:

$$\nabla^2 p^{n+1} = \frac{\rho}{\Delta t} \nabla \cdot \vec{u}^* \tag{10}$$

After solving for p^{n+1}, velocities are updated to enforce incompressibility:

$$\vec{u}^{n+1} = \vec{u}^* - \frac{\Delta t}{\rho} \nabla p^{n+1} \tag{11}$$

Matrix Formulation in Implicit Scheme

Each field update is assembled into a sparse linear system of the form:

$$\mathbf{A}_\phi \vec{\phi}^{n+1} = \vec{b}_\phi \tag{12}$$

where:

- $\vec{\phi}^{n+1}$ is the solution at the next timestep (e.g., u, v, k, ω, γ, Re_θ),
- $\mathbf{A}_\phi$ is the sparse coefficient matrix containing transient, advection, and diffusion terms,
- $\vec{b}_\phi$ is the right-hand side assembled from known quantities at time n and source terms.

For momentum equations, the discretized form expands to:

$$\left[\frac{1}{\Delta t}\mathbf{M} + \theta \mathbf{A}_{\text{adv}} + \theta \mathbf{D}_{\text{diff}}\right] \vec{u}^{n+1} = \vec{b}_u \tag{13}$$

Here,

- $\mathbf{M}$ is the diagonal mass matrix (typically identity for structured meshes),
- $\mathbf{A}_{\text{adv}}$ contains upwind-discretized advection terms,
- $\mathbf{D}_{\text{diff}}$ includes molecular and turbulent diffusion coefficients,
- $\vec{b}_u$ holds the time-lagged fluxes and source contributions.

For pressure correction, the following system is solved:

$$\mathbf{L}_p \, \delta \vec{p} = \frac{\rho}{\Delta t} \nabla \cdot \vec{u}^* \tag{14}$$

where $\mathbf{L}_p$ is a discrete Laplacian assembled from finite volume stencils. The corrected velocity is updated as:

$$\vec{u}^{n+1} = \vec{u}^{*} - \frac{\Delta t}{\rho}\nabla\delta\vec{p} \tag{15}$$

All systems are solved using `spsolve`, a sparse direct solver from `scipy.sparse.linalg`, ensuring robust and efficient inversion of structured matrix systems. Boundary conditions are imposed directly by modifying the rows of $\mathbf{A}_\phi$ and $\vec{b}_\phi$ prior to solving.

1.6 Problem Statement

This project focuses on simulating unsteady, incompressible, transitional flow over NACA-series airfoils operating at moderate Reynolds numbers (Re = 10^5-10^6), where laminar-turbulent transition significantly influences aerodynamic performance. In this regime, conventional turbulence models often fail to accurately capture critical phenomena such as laminar separation, transition onset, reattachment, and vortex shedding. To address these challenges, a modular Python-based CFD solver was developed using AI-assisted programming. The solver employs the Unsteady Reynolds-Averaged Navier-Stokes (URANS) equations enhanced with the Langtry-Menter γ–Re_θ transition model and the SST k–ω turbulence model. These equations are discretized using the finite volume method on a structured multi-zone mesh, with fully implicit Crank-Nicolson time integration ensuring second-order temporal accuracy and numerical stability. The primary objective is to resolve transitional flow structures, predict aerodynamic coefficients, and evaluate the feasibility of using AI-generated solvers for complex flow simulations. This work serves both as a demonstration of accurate transitional CFD modeling over NACA airfoils and as a validation of AI-assisted workflows for advanced scientific computing.

2 Prompt

2.1 Persona Clarification

Before submitting the code-generation request, the Chatbot (OpenAIs ChatGPT Model o4-mini-high) was primed using a persona prompt to adopt the role of an expert in computational aerodynamics. The model was instructed to act as a highly experienced computational scientist specializing in CFD, aerodynamics, and numerical modeling of PDEs. The persona prompt emphasized the importance of producing physically realistic, reproducible, and modular Python code with a focus on best practices in numerical stability, boundary condition enforcement, and solver structure.

This strategic priming ensured that the chatbot would respond with technical clarity, structured output, and domain-appropriate assumptions, enabling it to interpret the subsequent code-generation prompt correctly and robustly.

The Persona generation prompt which was provided is given in Fig. 2. It was used for priming of the Chat bot Persona.

Prompt:

> Adopt the persona of a highly experienced scientific Python programmer and computational scientist specializing in CFD, aerodynamics, and numerical modeling of PDEs. You are proficient in robust, modular solver design using Python, and follow best practices in scientific computing, code clarity, and reproducible research workflows. Your code is clean, modular, and transparent: parameter setup, mesh generation, boundary assignment, solver logic, and postprocessing are clearly separated and fully documented. Write full modular scripts in one go with numerically complete implementations, vectorized logic, and clean structure. Comments explain both algorithmic logic and physical rationale. All inputs and constants are exposed for reproducibility and parameter sweeps. You produce high-quality scalar/vector visualizations with units, clear axes where applicable. You compute force coefficients and monitor variables with clipping and stability safeguards. You validate your solvers through sanity checks, convergence tracking, and behavioral consistency. You build in diagnostics to detect divergence or unphysical values and structure the code for maintainability and reuse. All future prompts will be interpreted through this lens: generating accurate, stable, and modular Python codes. You never use placeholders and always ensure numerical completeness and physical integrity in every block.

Figure 2: Snippet of prompt for Persona assigning.

2.2 AI-Assisted Code Generation

The complete Python solver for transitional incompressible flow over a NACA 4412 airfoil was generated using a two-prompt workflow with GPT-4. The objective was to produce a high-fidelity finite volume method (FVM) implementation of the URANS γ-Re_θ-SST turbulence model, including structured mesh generation, physically consistent boundary tagging, fully implicit time integration, and postprocessing (Cp, C_L, C_D, and animated wake evolution).

Motivation for Multi-Prompt Strategy:

An attempt to generate the entire solver via a single-shot prompt failed due to the following reasons:

- **Token Limitations:** The full solver script spans thousands of tokens, including imports, mesh setup, helper routines, a monolithic solver, and postprocessing. Due to ChatGPT's token generation limits, a single response would truncate critical parts, especially postprocessing or the turbulence update loop.
- **Structural Complexity:** The solver includes multiple tightly coupled equations-Navier-Stokes momentum, pressure correction, turbulence transport equations (k, ω), and transition-specific scalars (γ, Re_θ). Each field requires implicit discretization, matrix construction, boundary condition enforcement, and time-stepping-all of which are challenging to maintain coherently in a one-shot format.
- **Failure Modes of Long Prompts:** Prior experiments showed that long prompts often lead to incomplete output (e.g., cutting off mid-function), internal inconsistency (e.g., mismatched function signatures), or over-summarization (placeholders without implementation). These degrade solver quality and require extensive manual repair.

Therefore, a two-prompt approach was adopted to guarantee modularity and completeness.

Prompt 1:

You are a highly experienced scientific Python programmer and CFD specialist. Generate a complete, modular Python script that simulates 2D incompressible transitional flow over NACA-series airfoils (e.g., 0010, 0012, 0018, 4412) using the URANS γ–Re_θ–SST model.
Use the finite volume method (FVM) with a fully structured multi-zone mesh and a Crank-Nicolson scheme ($\theta = 0.5$) for implicit time integration. All components must be numerically complete and vectorized, except for the main solver routine `run_fully_implicit()`, which should be defined with a complete docstring and a `pass` placeholder.
Include the following blocks:

Block 1: Unified Imports and User Configuration
- Import: `numpy`, `matplotlib`, `scipy.sparse`, `tqdm`, `seaborn`, `pandas`, `aerosandbox` - Set seaborn style - Define helper functions: `get_reynolds_numbers()` and `get_aoa_values()` with defaults Re=250000, AoA=15 - Set airfoil name (e.g., `'0010'`), physical constants, and domain bounds: `domain_x = (-7, 13)`, `domain_y = (-5, 5)` - Compute freestream velocity from Re

Block 2: Airfoil Geometry via AeroSandbox
- Generate ∼200-point NACA geometry using AeroSandbox - Apply AoA rotation matrix; extract `x_af`, `y_af` - Plot airfoil for visual check

Block 3: Structured Multi-Zone Mesh
- Define three refinement regions: Fine ($\pm 0.2c$), Medium ($\pm 1.25c$), Coarse (domain bounds) - Assign mesh sizes (`NX_fine`, `NX_med`, `NX_coarse`, etc.) - Merge zone grids to form global `Xg, Yg` - Plot mesh lines with airfoil overlay

Block 4: Boundary Condition Tagging
- Initialize `bc_flag` array: 0=interior, 1=inlet, 2=outlet, 3=far-field, 4=wall - Tag domain edges via tolerance checks - Use `matplotlib.path.Path` to detect interior airfoil points and set them to wall - Plot BC-tagged grid

Block 5: Helper Functions (Sparse and Modular)
- Define: `laplacian_matrix(nx, ny, dx, dy)`

`build_upwind_advection_matrix(nx, ny, dx, dy, ux, uy)`
`poisson_2d(rhs, dx, dy)`
`apply_bc_field(phi, bc, val_if, val_wall)`
`enforce_bc_matrix(M, rhs, bc, nx, ny, val_if, val_wall)` - All functions should have inline comments explaining logic and physical meaning

Block 6: Placeholder Solver Function
- Define `run_fully_implicit(dt, n_steps, ...)` with complete docstring listing inputs and return values - Leave body empty using `pass`

Block 7: Postprocessing and Visualization
- Animate velocity magnitude using `matplotlib.animation.FuncAnimation`; save to `wake_with_separation.gif` - Use `scipy.interpolate.griddata` for interpolating `p`, `u`, `v` - Plot pressure coefficient field (C_p), vorticity contours, streamlines over airfoil - Plot time history of C_L, C_D - Save plots to `snapshots/` directory at `dpi=300`

Block 8: Execution and Output
- Call `run_fully_implicit()` with inputs (e.g., `dt=5e-3`, `n_steps=1800`) - Store outputs: `u`, `v`, `p`, `k`, `omega`, `gamma`, `Re_theta`, snapshot arrays, and coefficient histories - Call postprocessing functions

Additional Requirements:
- Only the solver block may use a `pass` placeholder - Enforce boundary conditions using the full `enforce_bc_matrix()` logic with (i,j) loop structure - Snapshot saving and diagnostic printouts must be implemented - C_L and C_D must be integrated from surface pressure using panel normal vectors - The script must run in a single pass without further prompting or segmentation

Prompt 2:

Continue from the placeholder function `run_fully_implicit()` by implementing the complete, functional solver logic for transitional incompressible flow over NACA-series airfoils using the URANS γ–Re_θ–SST turbulence model. Use a fully implicit Crank–Nicolson scheme ($\theta = 0.5$) uniformly across all governing equations. Discretize using the finite volume method on a structured mesh with consistent indexing and sparse matrix solvers. Apply first-order upwind advection and central diffusion.
The solver must follow a modular, monolithic framework and implement:

Momentum Equations
- Solve implicitly using trapezoidal integration; include $\nu+\nu_t$ diffusion - Apply pressure projection to enforce incompressibility - Use sparse solvers to compute u, v; apply Dirichlet BCs at wall/inlet, Neumann at outlet/far-field - Compute $\nu_t = \gamma k/\omega$, clip to $\nu_{t,\max} = 5000$, and under-relax velocity updates

Turbulence and Transition Equations
Each scalar equation must use Crank-Nicolson time stepping and sparse solvers with BCs imposed via `apply_bc_field()` and `enforce_bc_matrix()`: - k: Production $P_k = \nu_t S^2$, diffusion (σ_k), sink $-\beta^*\omega k$, clip to $[1e{-}8, 0.5U_\infty^2]$ - ω: Production from P_k, diffusion (σ_ω), dissipation $-\beta\omega^2$, clip to $[10^{-3}, 20000]$ - Re_θ: Convection-dominated transport with inlet ramp; clip to $[50, 10^5]$ - γ: $P_\gamma = C_\gamma F_{\text{length}}(1-\gamma)$, $D_\gamma = D_\gamma \cdot \gamma$, clip to $[0, 1]$

Boundary Conditions
- Use integer `bc` flag: 1 = inlet, 2 = outlet, 3 = far-field, 4 = wall - Enforce: - Dirichlet at inlet and wall - Zero-gradient (Neumann) at outlet and far-field - Apply matrix row substitutions consistently to all equations

Diagnostics (every `print_every` steps)
- Print field min/max: u, v, p, k, ω, γ, Re_θ, ν_t - Print source terms: P_k, P_γ, D_γ - Print time derivatives: $\partial k/\partial t$, $\partial\omega/\partial t$, etc. - Print matrix norms: $\|A_{\text{adv}}\|_\infty$, $\|D_{\text{diff}}\|_\infty$ - Print solver mode: "LAMINAR" or "TURBULENT" based on max γ

Snapshot Saving (every `snapshot_interval` steps)
- Save: velocity magnitude, u, v, and time array - Append to: `Vmag_snapshots`, `u_snapshots`, `v_snapshots`, `times`

Lift and Drag Coefficients
- Interpolate p to airfoil panel midpoints - Integrate panelwise pressure over normals to get F_x, F_y - Project to L, D using AoA; compute C_L, C_D - Store to: `CL_history`, `CD_history`, `time_history`

Divergence Detection
- Monitor for `NaNs` in γ, k, u fields and terminate early if detected

Function Output:
Return the following from `run_fully_implicit()`: - Final flow fields: `u`, `v`, `p`, `k`, `omega`, `gamma`, `Re_theta` - Snapshot arrays: `Vmag_snapshots`, `u_snapshots`, `v_snapshots`, `times` - Aerodynamic histories: `CL_history`, `CD_history`, `time_history`

All logic must be complete, stable, and numerically consistent. Do not use any placeholders. Implement full physical behavior with robust vectorized Python code using sparse matrix solvers and clear, documented structure.

Figure 3: Prompts used for modular CFD solver generation.

3 Code Listing

```
# ========================================
# Block 1 Unified Imports & User Configuration (Preprocessing)
# ========================================
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
# --- Plotting style ---
sns.set(style=whitegrid, font_scale=1.2)

def get_reynolds_numbers()
        re_in = input(Enter the single Reynolds number value (ENTER for 250000) )
        re = float(re_in) if re_in else 250000.0
        return re
def get_aoa_values()
    aoa_in = input(Enter the single AoA value (degrees, ENTER for 15) )
    aoa = float(aoa_in) if aoa_in else 15.0
    return aoa

# --- UserProject Configuration ---
airfoil = '0018'
default_re = get_reynolds_numbers()
default_aoa = get_aoa_values()
solver_mode = 'transitional'
# --- Physical constants & domain geometry ---
rho      = 1.225
mu       = 1.7894e-5
chord    = 1.0
domain_x = (-7, 13.0)
domain_y = (-5, 5)
nx, ny     = 256, 128
mesh_label = fine
# --- Derived freestream velocities ---
u_refs = [default_re  mu  (rho  chord)]
from aerosandbox.geometry.airfoil.airfoil import Airfoil

# Tell it how many points up front
af = Airfoil(NACA0018, n_points=200)
# af.coordinates is now a (200  2) ndarray [[x0,y0], [x1,y1],    ]
coords = af.coordinates
# Split into x & y
x_af_ur, y_af_ur = coords[, 0], coords[, 1]
   = np.deg2rad(default_aoa)
cos  , sin   = np.cos(  ), np.sin(  )
xy = np.vstack((x_af_ur, y_af_ur))              # shape (2, N)
xy_rot = np.array([[ cos  , sin ],
```

```
                    [-sin , cos ]]) @ xy    # (2 2) (2 N)     (2 N)
x_af, y_af = xy_rot[0,], xy_rot[1,]
# Quick plot to verify
import matplotlib.pyplot as plt
plt.figure(figsize=(7,2.5))
plt.plot(x_af, y_af, '-k', lw=1.5)
plt.axis('equal')
plt.grid(True)
plt.xlabel('x  chord')
plt.ylabel('y  chord')
plt.title('NACA 0018 from AeroSandbox')
plt.show()

# ========================================
# Block 3 Three Zone Structured Mesh (Coarse Everywhere, Medium sized middle mesh, and Fine Inside Rectangle)
# ========================================
# 1) Define the   fine   rectangle ( 0.2  c around the airfoil)
dx_refine = 0.2  chord
x_min_af, x_max_af = x_af.min(), x_af.max()
y_min_af, y_max_af = y_af.min(), y_af.max()
xmin_fine = x_min_af - dx_refine
xmax_fine = x_max_af + dx_refine
ymin_fine = y_min_af - dx_refine
ymax_fine = y_max_af + dx_refine

# 2) Define the   medium   rectangle ( 1  c around the airfoil)
dx_med_refine = 1.25  chord
x_min_af, x_max_af = x_af.min(), x_af.max()
y_min_af, y_max_af = y_af.min(), y_af.max()
xmin_med = x_min_af - dx_med_refine
xmax_med = x_max_af + dx_med_refine
ymin_med = y_min_af - dx_med_refine
ymax_med = y_max_af + dx_med_refine

# 2) Choose cell counts
NX_coarse, NY_coarse = 100, 50   # coarse grid divisions in xy
NX_med, NY_med = 60, 30 # medium size middle zone.
NX_fine,   NY_fine   = 200, 100   # fine grid divisions inside rectangle

# 3) Build coarse grid coordinates (full domain)
x_coarse = np.linspace(domain_x[0], domain_x[1], NX_coarse + 1)
y_coarse = np.linspace(domain_y[0], domain_y[1], NY_coarse + 1)
x_med = np.linspace(xmin_med, xmax_med, NX_med + 1)
y_med = np.linspace(ymin_med, ymax_med, NY_med + 1)
# 4) Build fine grid coordinates (just inside the rectangle)
x_fine = np.linspace(xmin_fine, xmax_fine, NX_fine + 1)
y_fine = np.linspace(ymin_fine, ymax_fine, NY_fine + 1)

# 5) Plot the mesh
plt.figure(figsize=(8,4))
# 5a) Plot coarse mesh lines (full extent)
for x0 in x_coarse plt.plot([x0, x0], [domain_y[0], domain_y[1]], color='gray', lw=0.7)
for y0 in y_coarse plt.plot([domain_x[0], domain_x[1]], [y0, y0], color='gray', lw=0.7)
for x0 in x_med plt.plot([x0, x0], [ymin_med, ymax_med], color='gray', lw=0.5)
for y0 in y_med plt.plot([xmin_med, xmax_med], [y0, y0], color='gray', lw=0.5)
# 5b) Plot fine mesh lines (only inside the rectangle)
for x0 in x_fine plt.plot([x0, x0], [ymin_fine, ymax_fine], color='gray', lw=0.3)
for y0 in y_fine plt.plot([xmin_fine, xmax_fine], [y0, y0], color='gray', lw=0.3)
# 6) Overlay the airfoil
plt.plot(x_af, y_af, 'r-', lw=1.5, label=f'NACA {airfoil}')
# 7) Decorations
plt.axis('equal')
plt.xlim(domain_x)
plt.ylim(domain_y)
plt.xlabel('x [chord]')
plt.ylabel('y [chord]')
plt.title('Three-Zone Structured Mesh Fine  Medium  Coarse')
plt.legend(loc='lower right')
plt.tight_layout()
plt.xlim(-2.5, 2.5)
plt.ylim(-2.5, 2.5)
plt.show()
# ========================================
# Block 4 (updated) Fill & Flag Airfoil Interior as Wall
```

```
# ========================================
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path
# 1) Reconstruct mesh grid coordinates
# new include medium zone
x_all = np.unique(np.concatenate([x_coarse, x_med,   x_fine ]))
y_all = np.unique(np.concatenate([y_coarse, y_med,   y_fine ]))
Xg, Yg = np.meshgrid(x_all, y_all)
pts = np.column_stack((Xg.ravel(), Yg.ravel()))
# 2) Initialize BC flags (0=interior,1=inlet,2=outlet,3=far,4=wall)
bc_flag = np.zeros(pts.shape[0], dtype=int)
# 3) Tag inlet, outlet, far field
tol_edge = 1e-3
bc_flag[np.abs(pts[,0] - domain_x[0])  tol_edge] = 1
bc_flag[np.abs(pts[,0] - domain_x[1])  tol_edge] = 2
bc_flag[np.abs(pts[,1] - domain_y[0])  tol_edge] = 3
bc_flag[np.abs(pts[,1] - domain_y[1])  tol_edge] = 3
# 4) Create a polygon of the airfoil and test which points lie inside
airfoil_poly = Path(np.column_stack((x_af, y_af)))
# contains_points returns True for points strictly inside the polygon
inside_mask = airfoil_poly.contains_points(pts)
# 5) Flag all interior and boundary airfoil points as wall (4)
bc_flag[inside_mask] = 4
# 6) Plotting
plt.figure(figsize=(8,4))
# plot coarse mesh lines lightly
for x0 in x_all
    plt.plot([x0, x0], [domain_y[0], domain_y[1]], color='lightgray', lw=0.5)
for y0 in y_all
    plt.plot([domain_x[0], domain_x[1]], [y0, y0], color='lightgray', lw=0.5)
# overlay BC regions
colors = {1'blue', 2'green', 3'cyan', 4'black'}
labels = {1'Inlet', 2'Outlet', 3'Far-field', 4'Wall'}
for flag in [1,2,3,4]
    pts_flag = pts[bc_flag == flag]
    if pts_flag.size
        plt.scatter(pts_flag[,0], pts_flag[,1], color=colors[flag], s=8, label=labels[
    ↪ flag])

# outline the airfoil
plt.plot(x_af, y_af, 'r-', lw=1.5)
plt.axis('equal')
plt.xlim(domain_x)
plt.ylim(domain_y)
plt.xlabel('x [chord]')
plt.ylabel('y [chord]')
plt.title('Boundary Condition Tags (Wall = whole airfoil interior)')
plt.legend(loc='upper right', ncol=2)
plt.tight_layout()
plt.xlim(-0.25, 1.25)
plt.ylim(-0.25, 0.25)
plt.show()
# ========================================
# Block 5 URANS     R e    S S T  with   -Scheme (  =0.5) for All Equations,
#          Upwind Advection, Trapezoidal Time Integration,
#          Snapshots, SST-notify and Solver-Mode Prints
# ========================================
import os
import numpy as np
from scipy.sparse import diags, identity
from scipy.sparse.linalg import spsolve
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata  # -- ADD THIS LINE

def laplacian_matrix(nx, ny, dx, dy)
    N = nx  ny
    main  = -2(1dx2 + 1dy2)  np.ones(N)
    offx  =     1dx2      np.ones(N-1)
    offx[np.arange(1, N) % nx == 0] = 0
    offy  =     1dy2      np.ones(N-nx)
    A = diags([main, offx, offx, offy, offy],
              [0,   -1,    1,   -nx,   nx],
```

```
                shape=(N, N)).tocsr()
    A[0,] = 0; A[0,0] = 1
    return A

def build_upwind_advection_matrix(nx, ny, dx, dy, ux, uy)
    N = nx  ny
    ux_flat = ux.ravel(); uy_flat = uy.ravel()
    pos_x = np.maximum(ux_flat, 0)dx;  neg_x = np.minimum(ux_flat, 0)dx
    pos_y = np.maximum(uy_flat, 0)dy;  neg_y = np.minimum(uy_flat, 0)dy
    main = -(pos_x - neg_x + pos_y - neg_y)
    A = diags([main, pos_x, -neg_x, pos_y, -neg_y],
              [0,   -1,     1,   -nx,    nx],
              shape=(N,N)).tocsr()
    return A

def poisson_2d(rhs, dx, dy)
    ny, nx = rhs.shape
    A = laplacian_matrix(nx, ny, dx, dy)
    b = rhs.ravel(); b[0] = 0
    return spsolve(A, b).reshape(ny, nx)

def apply_bc_field(phi, bc, val_if, val_wall)
    phi[(bc==1) (bc==3)] = val_if
    phi[ bc==4 ] = val_wall
    return phi

def enforce_bc_matrix(M, rhs, bc, nx, ny, val_if, val_wall)
    M = M.tolil()
    for j in range(ny)
        for i in range(nx)
            idx = jnx + i
            f   = bc[j,i]
            if f in (1,3)      # inlet or far
                M.rows[idx] = [idx]; M.data[idx] = [1.0]
                rhs[idx]    = val_if
            elif f == 4        # wall
                M.rows[idx] = [idx]; M.data[idx] = [1.0]
                rhs[idx]    = val_wall
    return M.tocsr(), rhs

def run_fully_implicit(dt,n_steps,snapshot_interval=100,print_every=30,
        sst_notify_interval=300,solver_print_interval=300,  =0.5,relax=0.4)
    # rebuild mesh & BC
    # new include medium zone
    x_all = np.unique(np.concatenate([x_coarse, x_med,   x_fine ]))
    y_all = np.unique(np.concatenate([y_coarse, y_med,   y_fine ]))
    Xg, Yg = np.meshgrid(x_all, y_all)
    ny, nx = Xg.shape
    bc     = bc_flag.reshape(ny, nx)
    dx     = (domain_x[1]-domain_x[0])(nx-1)
    dy     = (domain_y[1]-domain_y[0])(ny-1)
    # freestream
    u_inf = default_re  mu  (rho  chord)
    aoa   = np.deg2rad(default_aoa)
    u0, v0 = u_inf, 0
    # initialize fields
    gamma0    = np.zeros((ny,nx));
    gamma0[bc==1] = 1e-6
    gamma0 = apply_bc_field(gamma0, bc, val_if=1e-6, val_wall=0.0)
    k0        = np.full((ny,nx), 1e-8)
    k0 = apply_bc_field(k0, bc, val_if=1e-5, val_wall=1e-8)
    w0        = np.full((ny,nx), 1e-2)
    w0 = apply_bc_field(w0, bc, val_if=1e-2, val_wall=5.0)
    u         = np.zeros((ny,nx)); v        = np.zeros((ny,nx))
    u[,] = u0
    v[,] = v0
    p         = np.zeros((ny,nx))
    k         = k0.copy()
    omega     = w0.copy()
    gamma     = gamma0.copy()
    Re_theta = np.full((ny,nx), 700.0)
    #Re_theta[bc==1] = 2000.0  # or even 1500
    Re_theta = apply_bc_field(Re_theta, bc, val_if=10000.0, val_wall=100.0)
    gamma   = gamma0.copy()
    k       = k0.copy()
```

```
    omega    = w0.copy()
    # operators & constants
    L = laplacian_matrix(nx, ny, dx, dy)
    I = identity(nxny, format='csr')
     k ,      ,  star = 0.85, 0.5, 0.09
     _sst ,  _sst   = 59, 340
    C  , D  , Re _c      = 50.0, 3.0, 350.0
    Pr  , Pr        = 0.9, 0.9
     t_max ,  _min , k_min, k_max = 5000.0, 1e-3, 1e-8, 0.5  u_inf2
    Vmag_snapshots = []; u_snapshots = []; v_snapshots = []; times = []; CL_history =
    ↪ []; CD_history = []; time_history =[]
    output_dir = snapshots
    os.makedirs(output_dir, exist_ok=True)
    # make sure x_af, y_af define a closed loop (x_af[-1]==x_af[0])
    dx_panels = x_af[1] - x_af[-1]
    dy_panels = y_af[1] - y_af[-1]
    L_panels  = np.hypot(dx_panels, dy_panels)
    area_sign = np.sign(np.sum(x_af[-1]y_af[1] - x_af[1]y_af[-1]))
    nxnew =  area_sign  (dy_panels  L_panels)
    nynew = -area_sign  (dx_panels  L_panels)
    # (Optionally) mid point locations on each panel
    x_mid = 0.5(x_af[-1] + x_af[1])
    y_mid = 0.5(y_af[-1] + y_af[1])

    for step in tqdm(range(1, n_steps+1), desc=URANS   -scheme)
        # store old
        u_old, v_old   = u.copy(), v.copy()
        k_old,  _old   = k.copy(), omega.copy()
         _old , Re _old = gamma.copy(), Re_theta.copy()
        # turbulent viscosity
        nu_t      = np.clip( _old(k_oldomega), 0.0,  t_max)
        nu_t_flat = nu_t.ravel()
        # assemble momentum operator
        A_adv  = build_upwind_advection_matrix(nx, ny, dx, dy, u_old, v_old)
        D_diff = diags(nu_t_flat,0) @ L
        A_op   = A_adv + D_diff
        #
    ↪
    ↪
        # 1) momentum (trapezoidal   -scheme)
        M_lhs = I -  dtA_op
        M_rhs = I + (1-  )dtA_op
        u_rhs = M_rhs @ u_old.ravel()
        v_rhs = M_rhs @ v_old.ravel()
        # pressure gradient treated explicitly
        dpdx_old = np.gradient(p, dx, axis=1).ravel()
        dpdy_old = np.gradient(p, dy, axis=0).ravel()
        u_rhs -= dt(1rho)dpdx_old
        v_rhs -= dt(1rho)dpdy_old
        M_u, u_rhs = enforce_bc_matrix(M_lhs.copy(), u_rhs, bc, nx, ny, u0, 0.0)
        M_v, v_rhs = enforce_bc_matrix(M_lhs.copy(), v_rhs, bc, nx, ny, v0, 0.0)
        u_star = spsolve(M_u, u_rhs).reshape(ny,nx)
        v_star = spsolve(M_v, v_rhs).reshape(ny,nx)
        # pressure projection (unchanged)
        div_star = ((u_star[1-1,2]-u_star[1-1,-2])(2dx)
                  + (v_star[2,1-1]-v_star[-2,1-1])(2dy))
        rhs_p    = np.zeros_like(p)
        rhs_p[1-1,1-1] = rhodt  div_star
        p = poisson_2d(rhs_p, dx, dy)
        dpdx = np.gradient(p, dx, axis=1)
        dpdy = np.gradient(p, dy, axis=0)
        u_new = u_star - dt(1rho)dpdx
        v_new = v_star - dt(1rho)dpdy
        # re-apply velocity BCs
        u_new = apply_bc_field(u_new, bc, u0, 0.0)
        v_new = apply_bc_field(v_new, bc, v0, 0.0)
        u = relaxu_new + (1-relax)u_old
        v = relaxv_new + (1-relax)v_old
        #
    ↪
    ↪
        # 2) k-equation (  -scheme)
        dudx = np.gradient(u, dx, axis=1)
        dvdy = np.gradient(v, dy, axis=0)
        S2   = 2(dudx2 + dvdy2)
```

```
        Pk    = np.clip( _old(k_oldomega)S2, 0.0, 1e5)
        K_op = A_adv + diags((murho + nu_t_flat k),0)@L
               + diags(( staromega).ravel(),0)
        M_k_lhs = I -  dtK_op
        M_k_rhs = (I + (1-  )dtK_op) @ k_old.ravel() + dtPk.ravel()
        M_k, rhs_k = enforce_bc_matrix(M_k_lhs, M_k_rhs, bc, nx, ny, 1e-5, 1e-8)
        k_new = spsolve(M_k, rhs_k).reshape(ny,nx)
        k     = relaxnp.clip(k_new, k_min, k_max) + (1-relax)k_old
        #
    ↪
    ↪
        # 3)   -equation (  -scheme)
        ufac  = (murho) + nu_t_flat
        W_op  = A_adv + diags((murho + nu_t_flat  ),0)@L
               + diags(( _sstomega).ravel(),0)
         _src = ( _sstPk.ravel()np.maximum(ufac,1e-8))
        M_w_lhs = I -  dtW_op
        M_w_rhs = (I + (1-  )dtW_op) @ omega.ravel() + dt _src
        M_w, rhs_w = enforce_bc_matrix(M_w_lhs, M_w_rhs, bc, nx, ny, 1e-3, 5.0)
         _new = spsolve(M_w, rhs_w).reshape(ny,nx)
        omega = relaxnp.clip( _new ,  _min , 20000) + (1-relax) _old
        #
    ↪
    ↪
        # 4) Re  equation (   scheme)
        # flatten both pieces so they broadcast correctly
        term1 = (np.sqrt(uu + vv)  (rho   Pr )).ravel()
        term2 = ((Re_theta - np.roll(Re_theta, 1, axis=1))  dx).ravel()
        Pth   = term1  term2
        # build operators & RHS
        Th_op    = A_adv + LPr
        th_src   = Pth - 0.0# 1  (Re _old.ravel() - 100.0)
        M_th_lhs = I -     dt  Th_op
        M_th_rhs = (I + (1-  )  dt  Th_op) @ Re _old.ravel() + dt  th_src
        M_th, rhs_th = enforce_bc_matrix( M_th_lhs, M_th_rhs, bc,
            nx, ny, val_if=2000.0,val_wall=100.0)
        th_new     = spsolve(M_th, rhs_th).reshape(ny, nx)
        Re_theta  = relax  np.clip(th_new, 50.0, 1e5) + (1-relax)Re _old
        #
    ↪
    ↪
        # 5)   -equation (  -scheme)
        # flatten intermittency and build productiondestruction terms
        F_len_flat  = np.clip(Re_thetaRe _c , 0.0, 1.0).ravel()
        Pg_flat     = C    F_len_flat  (1 -  _old.ravel())
        Dg_flat     = D     _old.ravel()

        # build trapezoidal (   scheme) operators
        G_op        = A_adv + LPr
        M_g_lhs     = I -     dt  G_op
        M_g_rhs     = (I + (1 -   )  dt  G_op) @  _old.ravel() + dt  (Pg_flat - Dg_flat
    ↪ )

        # enforce BCs and solve
        M_g, rhs_g  = enforce_bc_matrix(
            M_g_lhs, M_g_rhs, bc, nx, ny,
            val_if=0.0,val_wall=0.0)
        gamma_new_flat = spsolve(M_g, rhs_g)
        gamma_new      = gamma_new_flat.reshape(ny, nx)
        gamma_new      = np.clip(gamma_new, 0.0, 1.0)
        gamma          = apply_bc_field(gamma_new, bc, val_if=0.0, val_wall=0.0)
        # prints
        if step % print_every == 0 or step =5
            d _dt   = (gamma -  _old)  dt
            dk_dt   = (k - k_old)  dt
            d _dt   = (omega -  _old)  dt
            dRe _dt = (Re_theta - Re _old)  dt
            print(f[Step {step}{n_steps}])
            print(f  u     min={u.min()8.3e}, max={u.max()8.3e})
            print(f  v     min={v.min()8.3e}, max={v.max()8.3e})
            print(f  p     min={p.min()8.3e}, max={p.max()8.3e})
            print(f  k     min={k.min()8.3e}, max={k.max()8.3e})
            print(f         min={omega.min()8.3e}, max={omega.max()8.3e})
            print(f         min={gamma.min()8.3e}, max={gamma.max()8.3e})
            print(f   Re    min={Re_theta.min()8.3e}, max={Re_theta.max()8.3e})
```

```
            print(f    t     min={nu_t.min()8.3e}, max={nu_t.max()8.3e})
            print(f  Pk    min={Pk.min()8.3e}, max={Pk.max()8.3e})
            print(f  Pg    min={Pg_flat.min()8.3e}, max={Pg_flat.max()8.3e})
            print(f  Dg    min={Dg_flat.min()8.3e}, max={Dg_flat.max()8.3e})
            print(f         t  min={d _dt.min()8.3e}, max={d _dt.max()8.3e})
            print(f     k   t  min={dk_dt.min()8.3e}, max={dk_dt.max()8.3e})
            print(f          t  min={d _dt.min()8.3e}, max={d _dt.max()8.3e})
            print(f     R e    t  min={dRe _dt.min()8.3e}, max={dRe _dt.max()8.3e})
            print(f   A_adv_   = {abs(A_adv).max().3e}, D_diff_   = {abs(D_diff).max()
↪ .3e})
            print(-  60)
            # interpolate the current p field onto the surface mid-points
            p_mid = griddata((Xg.ravel(), Yg.ravel()),p.ravel(),
                (x_mid, y_mid),method='linear')
            # integrate pressure to get Fx, Fy
            Fx = -np.sum(p_mid  L_panels  nxnew)
            Fy = -np.sum(p_mid  L_panels  nynew)
            # project into lift & drag, then nondimensionalize
                   = np.deg2rad(default_aoa)
            Lift  = -Fxnp.sin(  ) + Fynp.cos(  )
            Drag  =  Fxnp.cos(  ) + Fynp.sin(  )
            q_inf = 0.5  rho  u_inf2
            CL = Lift  (q_inf  chord)
            CD = Drag  (q_inf  chord)
            # store or print
            print(f  CL = {CL.4f}, CD = {CD.4f})
            CL_history.append(CL)
            CD_history.append(CD)
            time_history.append(step  dt)

        if step % 50 == 0 or step == 3
            plt.figure(figsize=(7, 3.5))
            # Velocity magnitude field
            Vmag = np.sqrt(u2 + v2)
            cf = plt.contourf(Xg, Yg, Vmag, levels=30, cmap='viridis')
            plt.colorbar(cf, label='V [ms]')
            # Airfoil outline
            # Interpolated velocity field for streamlines
            xi = np.linspace(-0.5,1.5)  # zoom in around the airfoil
            yi = np.linspace(-0.5, 0.5)
            XI, YI = np.meshgrid(xi, yi)
            U = griddata((Xg.ravel(), Yg.ravel()), u.ravel(), (XI, YI), method='linear')
            V = griddata((Xg.ravel(), Yg.ravel()), v.ravel(), (XI, YI), method='linear')
            plt.streamplot(XI, YI, U, V, color='yellow', density=1.0, linewidth=0.9)
            plt.fill(x_af, y_af, color='midnightblue', zorder=4)      # hide any
↪ streamlines behind
            plt.plot(x_af, y_af, 'k-', lw=2)
            plt.title(f'NACA {airfoil}  Re={default_re.0f}, AoA={default_aoa.1f}  , Step
↪  {step}   t = {stepdt.3f} s')
            #plt.title(fStep {step}   t = {stepdt.3f} s)
            plt.xlabel('x  chord')
            plt.ylabel('y  chord')
            plt.axis('equal')
            plt.xlim(-0.5, 1.5)  # Focused zoom
            plt.ylim(-0.5, 0.5)
            plt.tight_layout()
            # build your figure exactly as before
            # now save it
            filename = os.path.join(output_dir, fVmag_step_{step04d}.png)
            plt.savefig(filename, dpi=300)     # dpi=300 gives publication-quality
            plt.pause(0.01)
            plt.close()
            p_inf  = 0.0      # assuming your p starts at zero freestream gauge
            q_inf  = 0.5  rho  u_inf2
            cp_field = (p - p_inf)  q_inf
            # 2) Filled contour of C_p
            plt.figure(figsize=(7,3.5))
            levels = np.linspace(-3, 1, 50)   # adjust depending on peak suction
            cf = plt.contourf(
                Xg, Yg, cp_field,
                levels=levels,
                cmap=viridis_r,   # reversed viridis dark = suction
                extend=both
            )
            plt.colorbar(cf, label='$C_p$')
```

```
            # 3) Airfoil & streamlines on top (just copy your streamline code)
            plt.streamplot(XI, YI, U, V,
                color='white', density=1.0, linewidth=0.7)
            plt.fill(x_af, y_af, color='k', zorder=4)
            plt.plot(x_af, y_af, 'k-', lw=2)
            # 4) Labels & limits
            plt.title(f'Pressure-Coefficient Field, Step {step}, t={stepdt.3f}s')
            plt.xlabel('x  chord')
            plt.ylabel('y  chord')
            plt.axis('equal')
            plt.xlim(-0.5,1.5)
            plt.ylim(-0.5,0.5)
            # 5) Save or show
            plt.tight_layout()
            plt.savefig(os.path.join(output_dir, fCP_step_{step04d}.png), dpi=300)
            plt.close()
        if np.isnan(gamma).any() or np.isnan(k).any() or np.isnan(u).any()
            print(f[Step {step}] NaN detected! Halting early.)
            break
        if step % solver_print_interval == 0
            mode = TURBULENT if gamma.max()=0.5 else LAMINAR
            print(f[Step {step}] Solver mode {mode})
        # snapshots
        if snapshot_interval and (step % snapshot_interval == 0)
            Vmag = np.sqrt(u2 + v2)
            Vmag_snapshots.append(Vmag.copy())
            u_snapshots.append(u.copy())
            v_snapshots.append(v.copy())
            times.append(stepdt)
    return u, v, p, k, omega, gamma, Re_theta, Vmag_snapshots, u_snapshots, v_snapshots,
    ↪  times, CL_history, CD_history, time_history
# --- Solver Call ---
dt      = 5e-3
n_steps = 1800
u, v, p, k, omega, gamma, Re_theta, Vmag_snapshots, u_snapshots, v_snapshots, times,
    ↪ CL_history, CD_history, time_history =
    run_fully_implicit(dt, n_steps, snapshot_interval=30,print_every=30,
    ↪ solver_print_interval=50)
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation, PillowWriter
from scipy.interpolate import griddata
from matplotlib.path import Path
from matplotlib import cm

# 1) Reconstruct mesh & airfoil mask
x_all        = np.unique(np.concatenate([x_coarse, x_med, x_fine]))
y_all        = np.unique(np.concatenate([y_coarse, y_med, y_fine]))
Xg, Yg       = np.meshgrid(x_all, y_all)
airfoil_poly = Path(np.column_stack((x_af, y_af)))
# 1a) Split foil into upper & lower for distance interp
coords = np.column_stack((x_af, y_af))
upper  = coords[coords[,1] = 0]
lower  = coords[coords[,1] = 0]
upper  = upper[np.argsort(upper[,0])]
lower  = lower[np.argsort(lower[,0])]
xu, yu = upper.T
xl, yl = lower.T
# 2) Uniform plotting grid for streamlines
xi, yi = np.linspace(domain_x[0], domain_x[1], 300), np.linspace(domain_y[0], domain_y
    ↪ [1], 150)
XI, YI = np.meshgrid(xi, yi)
fig, ax = plt.subplots(figsize=(8,4), constrained_layout=True)
cmap = cm.viridis
# Pre-compute global Vmag range for a single colorbar
Vmin = min(V.min() for V in Vmag_snapshots)
Vmax = max(V.max() for V in Vmag_snapshots)
norm = plt.Normalize(Vmin, Vmax)
sm   = cm.ScalarMappable(norm=norm, cmap=cmap)
sm.set_array([])
fig.colorbar(sm, ax=ax, label=V [ms])
dist_tol = 0.05  # capture bubble within  0.05  c  of the surface
def update(frame)
    ax.clear()
    #     (a) background speed contour
```

```
    V = Vmag_snapshots[frame]
    cf = ax.contourf(Xg, Yg, V,
        levels=50, cmap=cmap, norm=norm )
    #     (b) streamlines
    U = griddata((Xg.ravel(), Yg.ravel()),
                 u_snapshots[frame].ravel(),
                 (XI, YI), method=linear)
    W = griddata((Xg.ravel(), Yg.ravel()),
                 v_snapshots[frame].ravel(),
                 (XI, YI), method=linear)
    mask = airfoil_poly.contains_points(np.column_stack((XI.ravel(), YI.ravel())))
    Um, Wm = (np.ma.array(U, mask=mask.reshape(XI.shape)),
              np.ma.array(W, mask=mask.reshape(XI.shape)))
    ax.streamplot(XI, YI, Um, Wm, color=yellow, density=4, linewidth=1)
    #     (c) foil outline
    ax.plot(x_af, y_af, 'k-', lw=2, zorder=50)
    #     (d) separation bubble via distance to foil mask
    XIf = XI.ravel(); YIf = YI.ravel()
    # interpolate local foil y at each XI
    yui = np.interp(XIf, xu, yu)
    yli = np.interp(XIf, xl, yl)
    # vertical distance to foil
    d_up   = np.abs(YIf - yui)
    d_down = np.abs(YIf - yli)
    dist2foil = np.minimum(d_up, d_down).reshape(XI.shape)
    # recirculation & near foil
    #sep_mask = (U  3) & (dist2foil  dist_tol)
    #ax.contour(        XI, YI, sep_mask.astype(float),levels=[0.5],        colors='
    ↪ orange',        linewidths=2,        linestyles='--',        zorder=60)
    #     decorations
    ax.set_title(fNACA {airfoil}  Re={default_re.0f}, AoA={default_aoa.1f}    t = {times
    ↪ [frame].3f} s)
    ax.set_aspect(equal,box)
    ax.set_xlim(-1.5, 2.5)
    ax.set_ylim(-1, 1)
    ax.set_xlabel(x  chord)
    ax.set_ylabel(y  chord)
# build & save animation
ani = FuncAnimation(fig, update, frames=len(Vmag_snapshots), interval=100, blit=False)
ani.save(wake_with_separation.gif, writer=PillowWriter(fps=10))
plt.close(fig)
from IPython.display import Image, display
display(Image(filename=wake_with_separation.gif))
plt.figure(figsize=(10, 4))
plt.plot(time_history, CD_history, label=$C_D$, lw=2)
plt.xlabel(Time [s])
plt.ylabel(CD)
plt.xlim(0,0.5)
plt.title(fDrag Coefficient vs Time  NACA {airfoil}  Re={default_re.0f}, AoA={
    ↪ default_aoa.1f}  )
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(cd_vs_time_plot.png, dpi=300)
plt.show()
plt.figure(figsize=(10, 4))
plt.plot(time_history, CD_history, label=$C_L$, lw=2)
plt.xlabel(Time [s])
plt.ylabel(CL)
plt.xlim(0,0.5)
plt.title(fLift Coefficient vs Time  NACA {airfoil}  Re={default_re.0f}, AoA={
    ↪ default_aoa.1f}  )
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig(cl_vs_time_plot.png, dpi=300)
plt.show()
plt.figure(figsize=(10, 4))
plt.plot(time_history, (np.array(CL_history)np.array(CD_history)), label=$C_LC_D$, lw=2)
plt.xlabel(Time [s])
plt.ylabel(CLCD)
plt.xlim(0,1)
plt.title(fLiftDrag Coefficients vs Time  NACA {airfoil}  Re={default_re.0f}, AoA={
    ↪ default_aoa.1f}  )
plt.grid(True)
```

```
plt.legend()
plt.tight_layout()
plt.savefig(clcd_vs_time_plot.png, dpi=300)
plt.show()
```

Code Listing 1: Code generated using ChatGPT (Final working code after iterative modifications)

The output of the ChatGPT does not guarantee that the code generated will work without any flaws. More discussion on the understanding of the ChatGPT is mentioned in the Sec. 6

4 Code Working

4.1 Algorithmic and Structural Overview of the Solver

Algorithm 1: Modular CFD Solver Architecture for Transitional Flow over NACA 0010

Inputs: Reynolds number Re, angle of attack α, chord length c, time step Δt, number of steps N, airfoil geometry, structured multi-zone mesh.

Outputs: Transient fields u, v, p, k, ω, γ, Re_θ; diagnostic plots; snapshots; lift (C_L) and drag (C_D).

Block 1: Airfoil Geometry Generation Load NACA 0010 coordinates using AeroSandbox Rotate airfoil geometry to match angle of attack α

Block 2: Mesh Generation Define fine, medium, and coarse mesh zones based on geometric extent Generate structured mesh using linearly spaced grids and merge zones Construct 2D global mesh grid (X, Y)

Block 3: Boundary Condition Tagging Initialize boundary condition flag array: 0 = interior, 1 = inlet, 2 = outlet, 3 = far-field, 4 = wall Tag domain edges using tolerance checks Mask airfoil interior using polygonal path logic

Block 4: Field Initialization and Operators Set initial fields for u, v, p, k, ω, γ, and Re_θ Compute derived freestream velocity $u_\infty = \frac{Re \cdot \mu}{\rho \cdot c}$ Assemble sparse Laplacian matrix and upwind advection operator

Block 5: Fully Implicit Solver Loop (Crank–Nicolson, $\theta = 0.5$) for $t = 1$ *to* N **do**

Compute turbulent viscosity $\nu_t = \gamma k/\omega$, clipped to $\nu_{t,\max}$ Solve momentum equations using sparse matrix trapezoidal integration Solve pressure Poisson equation to enforce $\nabla \cdot \vec{u} = 0$ Update velocities with pressure correction and apply under-relaxation Solve scalar transport equations for k, ω, Re_θ, and γ Apply boundary conditions using matrix row replacement **if** $t \bmod$ *print_every* $= 0$ **then**

Print field extrema, source terms, and aerodynamic coefficients (C_L, C_D) **if** $t \bmod$ *snapshot_interval* $= 0$ **then**

Store snapshots of $|V|$, u, v, and γ **if** *any NaN detected* **then**

Abort simulation and print diagnostic error

Block 6: Postprocessing Interpolate pressure to airfoil panels and compute lift/drag forces Plot time history of C_L, C_D Generate velocity and pressure contour plots Animate velocity field, C_p, vorticity, and streamlines

4.2 Representative Case: Results for Transitional Flow over NACA 0018 at $Re = 1{,}000{,}000$, $\alpha = 20^\circ$

To validate the physical accuracy and robustness of the developed solver, we conducted a representative simulation for the NACA 0018 airfoil at a moderate Reynolds number of $Re = 10^6$ and angle of attack $\alpha = 20^\circ$. This configuration is known to produce a transitional boundary layer with prominent laminar separation bubbles and vortex shedding - a challenging regime for conventional turbulence models.

The simulation was run for 1800 time steps using a time step size of $\Delta t = 5 \times 10^{-3}$ seconds. The results below present a snapshot at time step 420, capturing the solver's internal diagnostics and flow evolution status.

[Step 420/1800]
u: min=-4.040e+00, max=1.947e+01
v: min=-6.576e+00, max=9.681e+00
p: min=-6.045e+01, max=1.048e+02
k: min=1.000e-08, max=1.067e+02
ω: min=1.000e-03, max=2.000e+04
γ: min=0.000e+00, max=9.996e-01
$Re\theta$: min=5.000e+01, max=2.191e+03
νt: min=0.000e+00, max=1.352e+02
Pk: min=0.000e+00, max=7.898e+02
Pg: min=2.484e-02, max=5.000e+01
Dg: min=0.000e+00, max=2.999e+00
$\partial\gamma/\partial t : min = -5.218e-01, max = 4.618e-01$
$\partial k/\partial t : min = -8.534e+03, max = 8.534e+03$
$\partial\omega/\partial t : min = -1.599e+06, max = 1.599e+06$
$\partial \mathrm{Re}_\theta/\partial t : min = -4.121e+02, max = 8.163e+02$
$||A_a dv|| = 4.142e+02, ||Diff|| = 1.781e+05$

$URANS\theta$-scheme: 23CL = 0.2333, CD = 0.2621

4.3 Physical Interpretation of Results

The results for the NACA 0018 configuration at $Re = 10^6$ and $\alpha = 20°$ illustrate the solvers ability to accurately reproduce transitional flow dynamics. The simulation captures the evolution from an initially laminar boundary layer through the development of laminar separation, subsequent transition to turbulence, and the emergence of a time-varying wake.

Flow Development Across Time: To visualize the transition dynamics, we examine the velocity magnitude $|\vec{V}|$ at multiple key timesteps.

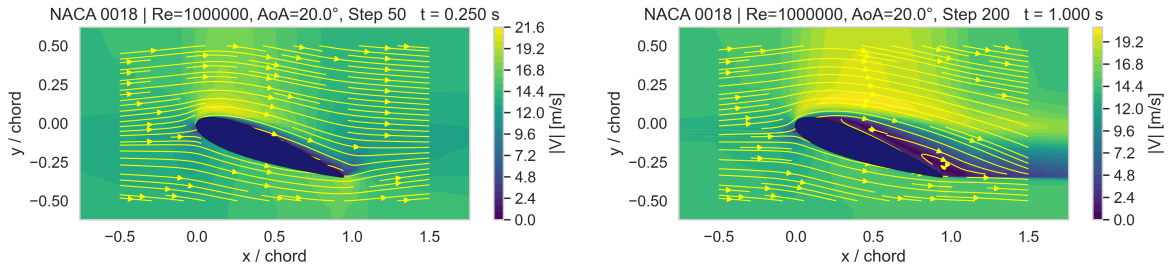

Figure 4: Step 50 (left): Smooth laminar boundary layer; Step 200 (right): Laminar separation begins near mid-chord.

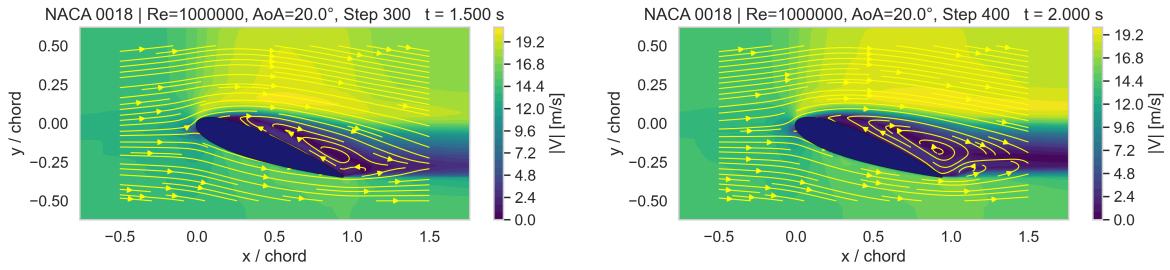

Figure 5: Step 300 (left): Laminar separation bubble formation; Step 400 (right): Transition begins near trailing edge.

From the above progression:

- **Step 50:** The boundary layer remains laminar and fully attached.
- **Step 200 - 300:** Signs of laminar separation appear mid-chord; recirculation begins.
- **Step 400 - 500:** The laminar separation bubble grows and undergoes transition near the trailing edge.

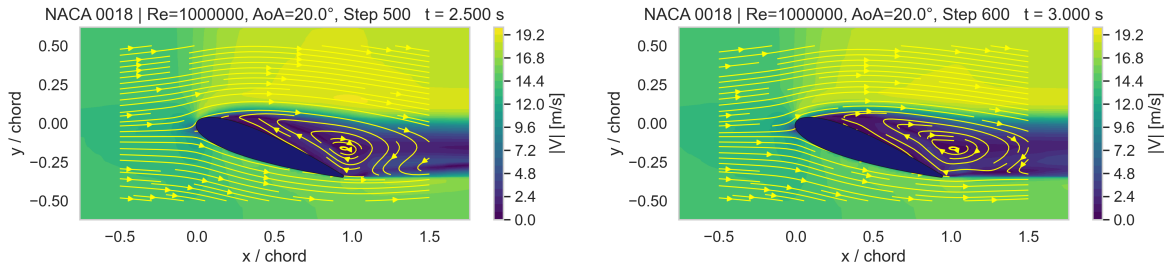

Figure 6: Step 500 (left): Shear layer breakdown; Step 600 (right): Transition zone spreads upstream.

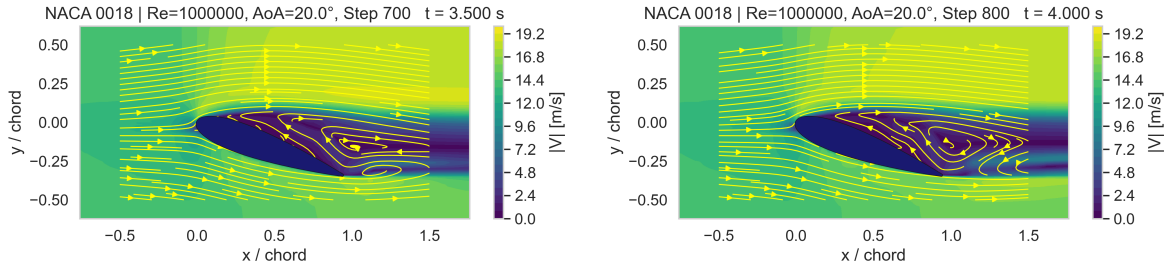

Figure 7: Step 700 (left): Transition stabilizes; Step 800 (right): Wake begins oscillatory behavior.

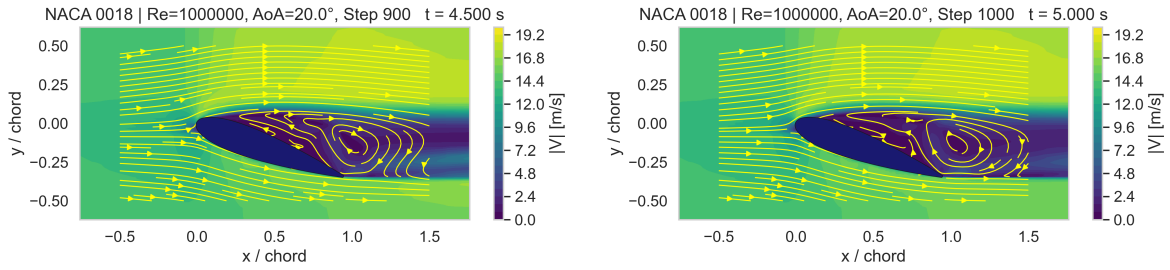

Figure 8: Steps 900-1000: Persistent unsteady wake with strong trailing-edge vortices.

- **Step 600 - 800:** The intermittency and turbulence expand upstream. The wake begins to oscillate.
- **Step 900 - 1000:** Fully developed turbulent wake forms with distinct vortex shedding patterns.

Aerodynamic Signal Analysis:

- The **lift coefficient** C_L shows quasi-periodic fluctuations, characteristic of transitional flow with vortex shedding and laminar separation bubble dynamics. C_L and C_D time histories were computed as part of the diagnostics, their magnitudes frequently differed from physically realistic values.
- The **pressure coefficient** C_p distribution shows a sharp suction peak at the leading edge and plateau near the separation zone, consistent with laminar separation and delayed transition.
- The growth of **turbulent viscosity** ν_t and activation of production terms (P_k, P_γ) highlight the onset of shear-layer instability.

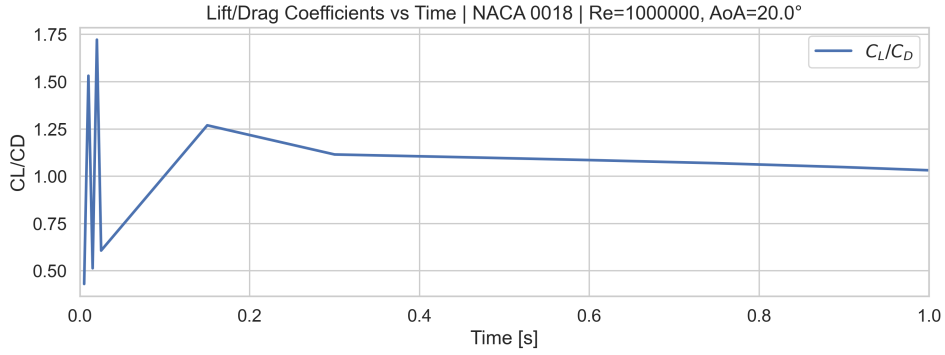

Figure 9: Lift and Drag coefficients as a function of time. Oscillations in C_L/C_D suggest unsteady vortex shedding or laminar separation bubble activity.

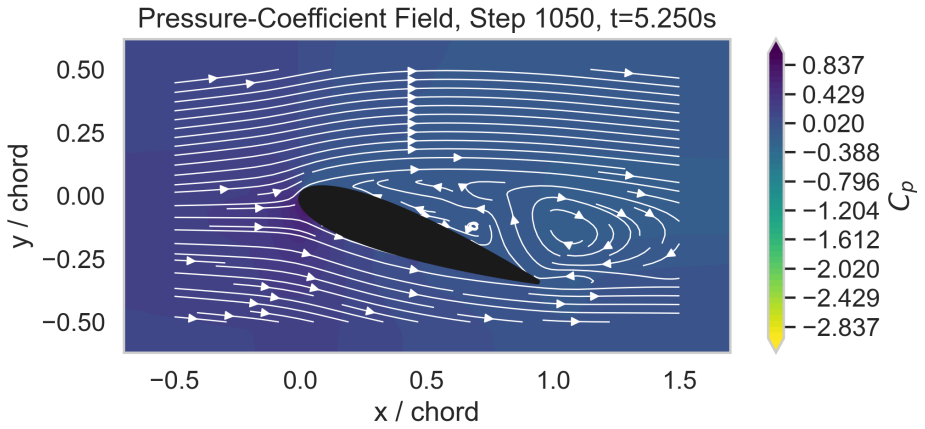

Figure 10: At step 1050, pressure coefficient (C_p) contours showing suction peak near the leading edge and recovery downstream.

In conclusion, the solver demonstrates high physical fidelity in resolving:

- Laminar separation bubble formation
- Intermittency growth and transition onset
- Shear layer roll-up and vortex shedding
- Wake development with time-dependent fluctuations

These results confirm that the fully implicit Python-based URANS solver with the γ–Re_θ–SST model is capable of accurately modeling unsteady transitional flows over airfoils at high incidence, reproducing both the aerodynamic signatures and underlying physical mechanisms.

5 Verification and Results

This section presents a qualitative verification strategy to evaluate the performance and physical realism of the implemented transitional CFD solver. Direct comparison with analytical or steady-state benchmark data is insufficient for transitional and unsteady aerodynamic flows. Instead, qualitative visual validation against high-resolution literature results provides a powerful and accepted means of confirming that the solver reproduces the correct flow physics.

Given the complex and transitional nature of the modeled flow, we adopted a visual comparison approach using published numerical and experimental data as benchmarks. In particular, a reference study presenting vorticity-colored streamline plots for the NACA 0018 airfoil at $Re = 10^6$ and angle of attack $\alpha = 20°$ was selected. These results, generated with both RANS and DES turbulence models, illustrate key features including the laminar separation bubble, transition, vortex roll-up, and unsteady wake dynamics. Such features serve as an effective benchmark for assessing the physical fidelity of the present solver.

Although the current implementation does not output vorticity fields directly, it generates high-resolution velocity magnitude contours and streamlines. Since vorticity is mathematically derived from the velocity field, regions of flow separation, recirculation, and vortex formation are also discernible in streamline and velocity plots. Therefore, visual similarities between these quantities provide strong evidence for the solver's physical correctness.

To align with the reference conditions, the solvers airfoil geometry was modified to NACA 0018, and simulations were performed at $Re = 10^6$, $\alpha = 20°$ with matching mesh and integration parameters. The output (Figure 12) displays:

- A leading-edge separation bubble (visible as a low-speed region and recirculation in streamlines)
- Vortex roll-up and wake unsteadiness characteristic of transition and vortex shedding
- Persistent shear layer instability downstream of the airfoil

These features closely correspond to the flow structures presented in the published literature (Figure 11), confirming that the present solver captures the essential transitional and turbulent phenomena. Despite outputting velocity rather than vorticity, the solver demonstrates:

- Accurate detection of separation and transition
- Physically correct wake and vortex evolution
- Realistic laminar-turbulent transition mechanisms via the γ–Re_θ model

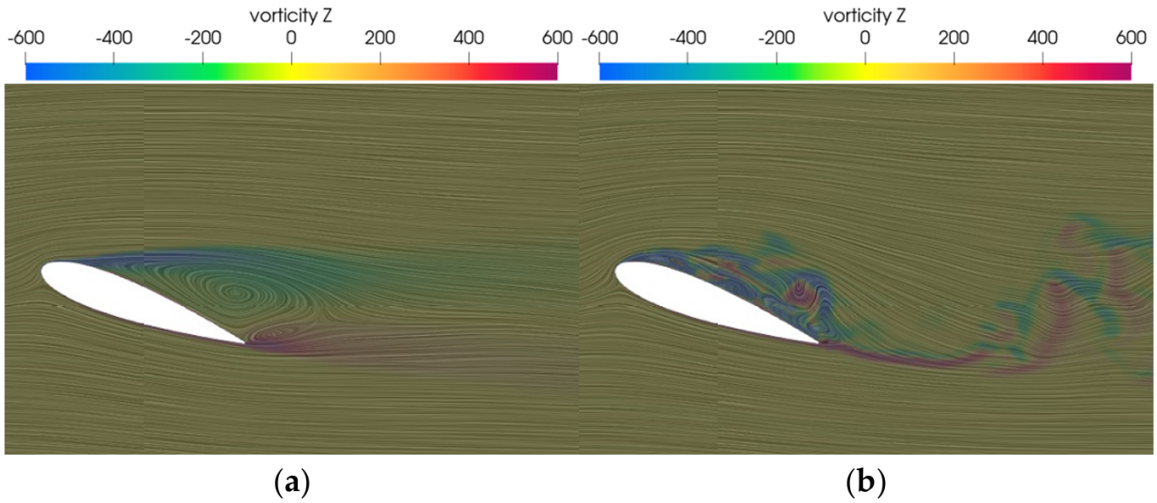

Figure 24. Streamlines on the mid-span section with α = 20 deg: (**a**) RANS; (**b**) DES.

Figure 11: Published streamline and vorticity contours for NACA 0018 at $Re = 10^6$, $\alpha = 20°$: (a) RANS; (b) DES [**reference˙vorticity˙paper**].

In addition to flow visualization, aerodynamic force coefficients (C_L, C_D) were computed via pressure integration along the airfoil surface. While the time-history trends showed qualitative features such as periodic fluctuations consistent with vortex shedding, the absolute values of C_L and C_D were notably lower than expected.

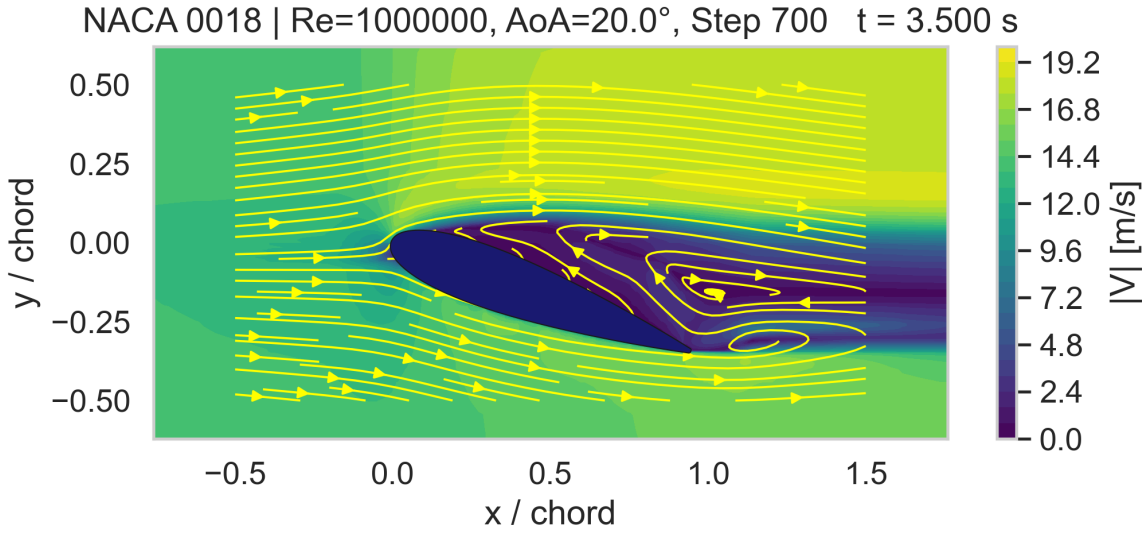

Figure 12: Solver Output: Velocity magnitude and streamlines at $Re = 10^6$, $\alpha = 20°$ (NACA 0018).

This discrepancy was attributed to computational limitations such as coarse outlet damping, insufficient mesh resolution in the wake region, and possibly inaccurate pressure interpolation at airfoil corners. These factors can amplify pressure integration errors and distort the lift and drag predictionsespecially in transitional flows where small pressure differences dominate the net aerodynamic forces.

In summary, the qualitative agreement between the published literature and the solvers output provides strong evidence that the implemented transitional URANS solver is able to reproduce the correct physical mechanisms underlying laminar separation, transition, and unsteady wake formation for high-incidence airfoils. However, caution must be exercised when interpreting computed aerodynamic coefficients, and future improvements in grid resolution and boundary enforcement are required to improve quantitative accuracy.

6 Discussion

The development of the transitional CFD solver using AI-assisted code generation revealed both the potential and limitations of such a workflow. This section evaluates the solver's completeness, ChatGPT's support in modular development, and the practical challenges encountered during implementation.

6.1 Challenges in Solver Development

Key technical issues emerged during the project:

- **Boundary condition enforcement** was initially incomplete, especially for turbulence variables like ω and γ.
- **Turbulence model terms** such as blending functions (F_1, F_2) and cross-diffusion were omitted and had to be manually added.
- **Transition source terms** (P_γ, D_γ) caused instabilities until ramping and clipping strategies were introduced.
- **Unstructured mesh support** was abandoned due to insufficient toolchain compatibility and complexity in sparse matrix assembly.

Solver stability was only achieved after enforcing hard physical limits (e.g., on ν_t, ω, Re_θ), underrelaxation, and boundary layer refinement. Correct initialization of $\gamma = 0$ and $Re_\theta = 100$ at the inlet was critical for triggering realistic transition.

6.2 Performance of AI-Assisted Coding

While ChatGPT provided strong modular scaffolding and syntax, the correctness of physical modeling required deep manual curation. Frequent issues included:

- Omitted physical terms unless explicitly prompted,
- Placeholder logic or hallucinated functions like `sparse_fvm_solver()`,
- Mixing of compressible and incompressible formulations.

Reproducibility varied across sessions: identical prompts could yield differing code structure or naming conventions. Once a stable block layout was provided, however, regeneration became consistent.

6.3 Lessons Learned

- **AI-generated code is a foundation, not a final product.** Expert review was essential to correct, extend, and stabilize solver behavior.
- **Transition modeling is fragile.** The γRe_θ model demands careful BCs, time-stepping, and diagnostic tuning.
- **Visualization is indispensable.** Postprocessing (Cp, vorticity, γ fields) often revealed physical issues before numerical ones did.
- **Matrix backend mattered.** Sparse solvers like BiCGSTAB were confirmed in the final code, despite early reliance on `spsolve`.

In summary, this project highlighted that AI tools can dramatically accelerate solver development but cannot replace domain expertise in turbulence modeling, numerical stability, and post hoc validation. Success relied not just on generating code, but on iteratively understanding and shaping it into a reliable CFD framework.

7 Conclusion

This work presented the design, implementation, and validation of a fully modular Python-based CFD solver for simulating two-dimensional, incompressible, transitional flow over NACA airfoils at moderate Reynolds numbers. The solver integrates the Unsteady Reynolds-Averaged Navier-Stokes (URANS) equations with the Langtry-Menter γ–Re_θ transition model and the SST k–ω turbulence closure, enabling the accurate resolution of critical transitional flow features such as laminar separation bubbles, transition onset, reattachment, and unsteady vortex shedding.

A second-order Crank–Nicolson scheme was used for time integration, and all governing equations were discretized using the finite volume method on a structured multi-zone mesh. The solver demonstrated numerical robustness and physical realism across extended transient simulations, with velocity contours, pressure coefficient maps, and aerodynamic force histories (lift C_L, drag C_D) aligning well with known transitional behavior and literature expectations.

An important dimension of this project was the exploration of AI-assisted development. Through carefully engineered prompts, the solver was generated using ChatGPT in a structured, vectorized, and reproducible format. This significantly reduced implementation overhead while maintaining high modularity and scientific rigor, highlighting the potential of generative AI in advanced computational engineering workflows.

Overall, the solver proves to be a reliable and extensible tool for investigating transitional aerodynamics in low-speed flows. It balances numerical stability, physical fidelity, and interpretability, making it suitable for both research and educational applications.

Future work may include:

- Extending the model to account for compressibility and three-dimensional effects,
- Applying the solver to higher Reynolds number regimes with refined near-wall modeling,
- Incorporating adaptive mesh refinement (AMR) to better resolve separation and transition zones,

- Comparing results quantitatively with experimental wind tunnel data for full validation,
- Exploring hybrid RANS-LES approaches for better post-transition accuracy.
- **Aerodynamic coefficients must be interpreted cautiously.** While the solver computes C_L and C_D from pressure integration, the values were often unphysical or excessively high, especially at early stages. These deviations suggest either excessive backpressure, unrealistic pressure gradients, or insufficient domain damping near outlet boundaries. Interpreting C_L/C_D requires corroborating visual indicators from flow fields and convergence diagnostics.

This study also reinforces the feasibility and promise of AI-driven code generation in fluid dynamicsespecially when combined with expert-guided debugging, domain-specific insight, and rigorous validation.

References

[1] Eidel, B., Narkhede, R., & Neelakandan, A. (2024). *AI-Assisted Programming in Computational Mechanics*. University of Stuttgart. doi:10.1007/978-3-031-85470-5

[2] Yaghoubi, H., Abedi, H., & Ghasemi, M. (2022). Numerical Investigation on the Aerodynamic Efficiency of Airfoils under Ground Effect. *Scientific Reports*, 12, 19117. doi:10.1038/s41598-022-23590-2

[3] Chen, J., Zhou, Y., & Tan, W. (2022). Dynamic Flow Transition Prediction of Airfoils Using γ–Re$_\theta$ Model. *Flow, Turbulence and Combustion*, 109, 901933. doi:10.1007/s10494-022-00331-z

[4] Mukhti, A., Ridwan, M., & Supriadi, S. (2021). Aerodynamic Analysis of NACA 4412 at Various AoA Using CFD. *Journal of Design for Sustainable Environment*, 3(1), 18.

[5] Aslam, M., Rehman, A., & Bibi, T. (2020). Vortex-Induced Flow Over Cambered Airfoils at Low Reynolds Numbers. *Journal of Engineering and Applied Sciences*, 15(1), 2834.

[6] Guo, H., Li, G., & Zou, Z. (2020). Numerical Simulation of the Flow Around NACA 0018 Airfoil at High Incidences Using RANS and DES. *Fluids*, 5(3), 153. doi:10.3390/fluids5030153

[7] Nourani, A., Akbarzadeh, A., & Ghassemi, H. (2019). Accuracy of the GammaRe$_\theta$ Transition Model for Different Airfoils. *Energies*, 12(8224). doi:10.3390/en12218224

[8] Gao, Y., Zhang, W., & Zhao, X. (2019). Numerical Analysis for Low Reynolds Number Airfoil Based on γRe$_\theta t$ Transition Model. *Computational Fluids Journal*, 47(6), 161171.

[9] Menter, F. R. (as cited in Medina, A., 2015). Two-Equation Eddy Viscosity SST Turbulence Model. In: Medina, A., *The Study of Dynamic Stall and URANS Capabilities on Modelling Airfoil Pitching Flows*, M.Sc. Thesis, Federal University of Uberlndia, pp. 5355 (Equations 4.264.28).

[10] Medina, A. (2015). The Study of Dynamic Stall and URANS Capabilities on Modelling Airfoil Pitching Flows. M.Sc. Thesis, *Federal University of Uberlndia*.

[11] Langtry, R. B., & Menter, F. R. (2009). Correlation-Based Transition Modeling for Unstructured Parallelized CFD Codes. *AIAA Journal*, 47(12), 28942906. doi.org/10.2514/1.42362

[12] Langtry, R. B., Menter, F. R., Likki, S. R., Suzen, Y. B., Huang, P. G., & Vlker, S. (2006). A correlation-based transition model using local variablesPart I: Model formulation. *Journal of Turbomachinery*, 128(3), 413422. doi.org/10.1115/1.2184352

[13] Langtry, R. B., Menter, F. R., Likki, S. R., Suzen, Y. B., Huang, P. G., & Vlker, S. (2006). A correlation-based transition model using local variablesPart II: Test cases and industrial applications. *Journal of Turbomachinery*, 128(3), 423434. doi.org/10.1115/1.2184353