ME – 285

Turbomachine Theory

Assignment 2

भारतीय विज्ञान संस्थान

Name: Sugesh Kumar S

Roll Number: 24050

Submission Date: Thursday 4th December, 2025

Department of Aerospace Engineering

Indian Institute of Science

Contents

1 Problem Statement

Develop a computer code to solve potential flow over a linear cascade of airfoils at an angle of attack using the **Conformal Mapping** technique combined with the **Vortex Panel Method**.

1.1 Objectives

1. Implement the conformal mapping $z_2 = \tanh(\frac{\pi z_1}{h})$ to transform an infinite linear cascade in the physical plane (z_1) into a single closed contour in the computational plane (z_2).

2. Solve the potential flow over the single body in the z_2-plane using a higher-order panel method.

3. Compute the aerodynamic properties, including the Pressure Coefficient (C_p) distribution and the flow field streamlines.

4. Visualize the geometry and flow in both the physical and transformed planes.

2 Theoretical Background

Solving potential flow directly for an infinite cascade is computationally expensive due to the need for infinite periodic singularities. A more efficient approach involves mapping the cascade to a single body.

2.1 Conformal Mapping

The core concept relies on the property that the region external to an infinite row of blades (linear cascade) in the physical z_1-plane can be mapped conformally onto the region external to a single closed contour in the z_2-plane. The specific transformation used is:

$$z_2 = \tanh\left(\frac{\pi z_1}{h}\right) \tag{1}$$

where h is the spacing between the blades in the cascade.

This hyperbolic tangent transformation maps the periodic strip of height h in the z_1-plane to the entire z_2-plane. Crucially, the transformation handles the boundaries at infinity as follows:

- **Upstream Infinity:** The region far upstream ($x_1 \to \infty$) maps to the singular point $(+1, 0)$ in the z_2-plane.
- **Downstream Infinity:** The region far downstream ($x_1 \to -\infty$) maps to the singular point $(-1, 0)$ in the z_2-plane.

2.2 Flow Singularities in Transformed Plane

Since the uniform flow at upstream and downstream infinity is mapped to finite points, the flow behavior must be reconstructed using point singularities:

- **At Inlet (+1, 0):** A source of strength Q and a vortex $\Gamma_{upstream}$. Here, $Q = V_1 h \cos(\alpha_1)$ represents the mass flow rate, and $\Gamma_{upstream} = V_1 h \sin(\alpha_1)$ represents the inlet tangential velocity.
- **At Outlet (-1, 0):** A sink of strength $-Q$ (by mass conservation) and a vortex $\Gamma_{downstream}$.

While Q and $\Gamma_{upstream}$ are known from the inlet boundary conditions, $\Gamma_{downstream}$ depends on the lift generated by the blades and is initially **unknown**. It must be solved for as part of the linear system.

2.3 Discretization

In the z_2-plane, the single transformed contour is discretized into M panels defined by $M + 1$ nodes. Unlike constant-strength source panels, this method employs a **higher-order vortex panel method** where the vorticity strength γ varies linearly across each panel.

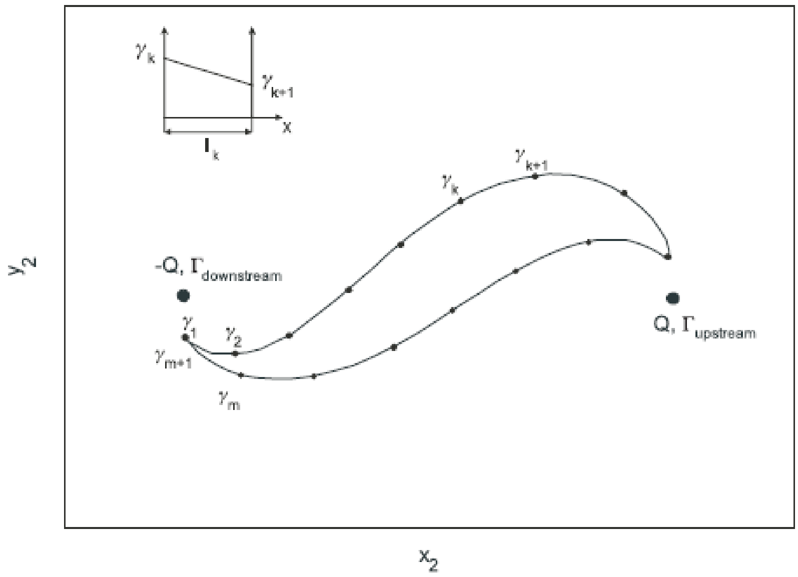

Figure 1: Discretization of the transformed single contour into vortex panels with linearly varying vorticity.

As shown in Figure 1, the vorticity at node k is γ_k and at node $k + 1$ is γ_{k+1}. The induction of velocity by these linearly varying vortex sheets is calculated analytically using influence coefficients.

3 Methodology and Implementation

The solution is implemented in Python using 'numpy' for matrix operations and 'matplotlib' for visualization.

3.1 Step 1: Geometry Generation

A NACA 0012 airfoil is generated. The coordinates are rotated by the stagger angle β and shifted to form the central blade of the cascade in the z_1-plane.

3.2 Step 2: Transformation

The blade coordinates are transformed to the z_2-plane using 'np.tanh'. This results in a distorted "S-shape" or loop (depending on solidity) representing the entire cascade.

3.3 Step 3: Matrix Assembly

We construct a system of linear equations $[A]\{\Gamma\} = \{RHS\}$ to solve for the unknown vortex strengths.

- **Unknowns:** There are $M + 1$ nodal vortex strengths $(\gamma_1, \gamma_2, \ldots, \gamma_{M+1})$ and one unknown downstream circulation parameter $\Gamma_{downstream}$. Total unknowns $= M + 2$.

The matrix is assembled based on the following physical conditions:

3.3.1 1. No-Penetration Condition (Rows 1 to M)

For each of the M panels, the velocity component normal to the surface must be zero at the panel control point (midpoint). The velocity at any control point is the sum of:

1. Velocity induced by all vortex panels (functions of unknown γ).
2. Velocity induced by the unknown point vortex $\Gamma_{downstream}$ at $(-1, 0)$.
3. Velocity induced by the known singularities $(Q, \Gamma_{upstream})$.

This yields M linear equations.

3.3.2 2. Kutta Condition (Row $M + 1$)

To ensure smooth flow at the trailing edge, the vorticity at the first node (γ_1) and the last node (γ_{M+1}) must cancel out (or sum to zero depending on winding convention). In this code:

$$\gamma_1 + \gamma_{M+1} = 0 \tag{2}$$

3.3.3 3. Circulation/Periodicity Condition (Row $M + 2$)

The change in tangential velocity between far upstream and far downstream is directly related to the total circulation around a blade. This provides the closure equation for $\Gamma_{downstream}$:

$$\Gamma_{blade} - \Gamma_{downstream} = \Gamma_{upstream} \tag{3}$$

where $\Gamma_{blade} = \sum \gamma_i \cdot s_i$ (integral of vorticity over the panels). This equation links the panel strengths to the downstream point vortex strength.

3.4 Step 4: Flow Field Reconstruction

A mesh grid is created in the z_2 plane. Velocities are calculated by summing contributions from all panels and the point singularities at ± 1. These velocities are then mapped back to the physical plane using the derivative of the mapping function $|V_1| = |V_2||\frac{dz_2}{dz_1}|$.

4 Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path

def get_naca_4_digit_airfoil(code, c=1.0, n_points=150):
    m = int(code[0]) / 100.0
    p = int(code[1]) / 10.0
    t = int(code[2:]) / 100.0

    beta = np.linspace(0, np.pi, n_points)
    x = c * (0.5 * (1 - np.cos(beta)))

    yt = 5 * t * c * (0.2969 * np.sqrt(x/c) - 0.1260 * (x/c)
                      - 0.3516 * (x/c)**2 + 0.2843 * (x/c)**3
                      - 0.1015 * (x/c)**4)

    yc = np.zeros_like(x)
    dyc_dx = np.zeros_like(x)

    for i in range(len(x)):
        if x[i] <= p * c:
            if p != 0:
                yc[i] = (m / p**2) * (2 * p * (x[i]/c) - (x[i]/c)**2)
                dyc_dx[i] = (2 * m / p**2) * (p - x[i]/c)
        else:
            if p != 1:
                yc[i] = (m / (1 - p)**2) * ((1 - 2 * p) + 2 * p * (x[i]/c) - (x[i]/c)**2)
                dyc_dx[i] = (2 * m / (1 - p)**2) * (p - x[i]/c)

    theta = np.arctan(dyc_dx)
    xu = x - yt * np.sin(theta); yu = yc + yt * np.cos(theta)
    xl = x + yt * np.sin(theta); yl = yc - yt * np.cos(theta)

    x_coords = np.concatenate((xl[::-1], xu[1:]))
    y_coords = np.concatenate((yl[::-1], yu[1:]))

    return x_coords, y_coords

def rotate_coords(x, y, angle_deg):
    theta = np.radians(angle_deg)
    xr = x * np.cos(theta) - y * np.sin(theta)
    yr = x * np.sin(theta) + y * np.cos(theta)
    return xr, yr

def solve_cascade_panel_method_R2L(blades_z2_nodes, V1, alpha_deg, spacing, chord):
    XB = blades_z2_nodes.real
```

```
    YB = blades_z2_nodes.imag
    M_nodes = len(XB)
    N_panels = M_nodes - 1

    X = 0.5 * (XB[:-1] + XB[1:])
    Y = 0.5 * (YB[:-1] + YB[1:])
    dx = XB[1:] - XB[:-1]; dy = YB[1:] - YB[:-1]
    S = np.sqrt(dx**2 + dy**2); theta = np.arctan2(dy, dx)

    alpha_rad = np.radians(alpha_deg)
    Q = V1 * spacing * np.cos(alpha_rad)

    Gamma_up = -V1 * spacing * np.sin(alpha_rad)

    CN1 = np.zeros((N_panels, N_panels)); CN2 = np.zeros((N_panels, N_panels))
    CT1 = np.zeros((N_panels, N_panels)); CT2 = np.zeros((N_panels, N_panels))

    print(f"  > Calculating Influence Matrix for {N_panels} panels...")
    for i in range(N_panels):
        for j in range(N_panels):
            if i == j:
                CN1[i,j] = -1.0; CN2[i,j] = 1.0; CT1[i,j] = 0.5*np.pi; CT2[i,j] = 0.5*np.pi
            else:
                A = -(X[i]-XB[j])*np.cos(theta[j]) - (Y[i]-YB[j])*np.sin(theta[j])
                B = (X[i]-XB[j])**2 + (Y[i]-YB[j])**2
                C = np.sin(theta[i]-theta[j]); D = np.cos(theta[i]-theta[j])
                E = (X[i]-XB[j])*np.sin(theta[j]) - (Y[i]-YB[j])*np.cos(theta[j])
                F = np.log(1.0 + (S[j]**2 + 2*A*S[j])/B)
                G = np.arctan2((E*S[j]), (B + A*S[j]))
                P = (X[i]-XB[j])*np.sin(theta[i]-2*theta[j]) + (Y[i]-YB[j])*np.cos(theta[i]-2*theta[j])
                Q_geom = (X[i]-XB[j])*np.cos(theta[i]-2*theta[j]) - (Y[i]-YB[j])*np.sin(theta[i]-2*theta[j])

                CN2[i,j] = D + (0.5*Q_geom*F)/S[j] - (A*C + D*E)*(G/S[j])
                CN1[i,j] = 0.5*D*F + C*G - CN2[i,j]
                CT2[i,j] = C + (0.5*P*F)/S[j] + (A*D - C*E)*(G/S[j])
                CT1[i,j] = 0.5*C*F - D*G - CT2[i,j]

    AN = np.zeros((N_panels, M_nodes)); AT = np.zeros((N_panels, M_nodes))
    for i in range(N_panels):
        AN[i, 0] = CN1[i, 0]; AN[i, -1] = CN2[i, -1]
        AT[i, 0] = CT1[i, 0]; AT[i, -1] = CT2[i, -1]
        for j in range(1, N_panels):
            AN[i, j] = CN1[i, j] + CN2[i, j-1]; AT[i, j] = CT1[i, j] + CT2[i, j-1]

    Num_Unknowns = M_nodes + 1
    A_sys = np.zeros((Num_Unknowns, Num_Unknowns))
    RHS_sys = np.zeros(Num_Unknowns)
```

```python

def get_vel_point_vortex(x_t, y_t, xv, yv):
    r2 = (x_t - xv)**2 + (y_t - yv)**2
    u = -(1.0 / (2*np.pi)) * (y_t - yv) / r2
    v =  (1.0 / (2*np.pi)) * (x_t - xv) / r2
    return u, v

def get_vel_point_source(x_t, y_t, xv, yv):
    r2 = (x_t - xv)**2 + (y_t - yv)**2
    u = (1.0 / (2*np.pi)) * (x_t - xv) / r2
    v = (1.0 / (2*np.pi)) * (y_t - yv) / r2
    return u, v

for i in range(N_panels):
    A_sys[i, 0:M_nodes] = AN[i, :]

    u_gd, v_gd = get_vel_point_vortex(X[i], Y[i], -1.0, 0.0)
    A_sys[i, M_nodes] = -u_gd*np.sin(theta[i]) + v_gd*np.cos(theta[i])

    u_stat = 0; v_stat = 0
    u, v = get_vel_point_source(X[i], Y[i], 1.0, 0.0); u_stat += Q * u; v_stat += Q * v
    u, v = get_vel_point_vortex(X[i], Y[i], 1.0, 0.0); u_stat += Gamma_up * u; v_stat += Gamma_up * v
    u, v = get_vel_point_source(X[i], Y[i], -1.0, 0.0); u_stat += -Q * u; v_stat += -Q * v
    RHS_sys[i] = -(-u_stat*np.sin(theta[i]) + v_stat*np.cos(theta[i]))

A_sys[N_panels, 0] = 1.0; A_sys[N_panels, M_nodes-1] = 1.0
for j in range(N_panels):
    A_sys[Num_Unknowns-1, j] += 0.5 * S[j]
    A_sys[Num_Unknowns-1, j+1] += 0.5 * S[j]
A_sys[Num_Unknowns-1, Num_Unknowns-1] = -1.0
RHS_sys[Num_Unknowns-1] = Gamma_up

print("  > Solving Linear System...")
Solution = np.linalg.solve(A_sys, RHS_sys)
gammas = Solution[0:M_nodes]; Gamma_down = Solution[M_nodes]

Vt_panels = np.zeros(N_panels); Cp_panels = np.zeros(N_panels)
z2_mid = X + 1j*Y
dz2_dz1 = (np.pi / spacing) * (1 - z2_mid**2)
map_scale = np.abs(dz2_dz1)

for i in range(N_panels):
    vt_induced = 0
    for j in range(M_nodes): vt_induced += AT[i, j] * gammas[j]
    u_gd, v_gd = get_vel_point_vortex(X[i], Y[i], -1.0, 0.0)
    vt_induced += Gamma_down * (u_gd*np.cos(theta[i]) + v_gd*np.sin(theta[i]))

    u_stat = 0; v_stat = 0
    u, v = get_vel_point_source(X[i], Y[i], 1.0, 0.0); u_stat += Q * u; v_stat += Q * v
    u, v = get_vel_point_vortex(X[i], Y[i], 1.0, 0.0); u_stat += Gamma_up * u; v_stat += Gamma_up * v
```

```
        u, v = get_vel_point_source(X[i], Y[i], -1.0, 0.0); u_stat += -
    Q * u; v_stat += -Q * v

        vt_static = u_stat*np.cos(theta[i]) + v_stat*np.sin(theta[i])
        Vt_z1 = (vt_induced + vt_static) * map_scale[i]
        Cp_panels[i] = 1 - (Vt_z1 / V1)**2

    return Cp_panels, z2_mid, gammas, Gamma_down, Q, Gamma_up

def compute_flow_field(X_grid, Y_grid, gammas, Gamma_down, Q, Gamma_up,
     blades_z2_nodes, spacing):
    z1_grid = X_grid + 1j * Y_grid
    z2_grid = np.tanh(np.pi * z1_grid / spacing)
    XB = blades_z2_nodes.real; YB = blades_z2_nodes.imag
    X_panel = 0.5 * (XB[:-1] + XB[1:])
    Y_panel = 0.5 * (YB[:-1] + YB[1:])
    dx = XB[1:] - XB[:-1]; dy = YB[1:] - YB[:-1]
    S = np.sqrt(dx**2 + dy**2)

    u2 = np.zeros_like(X_grid); v2 = np.zeros_like(Y_grid)

    for k in range(len(gammas)-1):
        rx = z2_grid.real - X_panel[k]; ry = z2_grid.imag - Y_panel[k];
     r2 = rx**2 + ry**2
        circ_k = gammas[k] * S[k]
        u2 += -(1.0/(2*np.pi))*ry/r2 * circ_k
        v2 +=  (1.0/(2*np.pi))*rx/r2 * circ_k

    def add_sing(x0, y0, str_val, is_src):
        rx = z2_grid.real - x0; ry = z2_grid.imag - y0; r2 = rx**2 + ry
    **2 + 1e-9
        if is_src: u2[:] += (str_val/(2*np.pi))*rx/r2; v2[:] += (
    str_val/(2*np.pi))*ry/r2
        else:      u2[:] += -(str_val/(2*np.pi))*ry/r2; v2[:] += (
    str_val/(2*np.pi))*rx/r2

    add_sing(1.0, 0.0, Q, True); add_sing(1.0, 0.0, Gamma_up, False)
    add_sing(-1.0, 0.0, -Q, True); add_sing(-1.0, 0.0, Gamma_down,
    False)

    dz2_dz1 = (np.pi/spacing) * (1 - z2_grid**2)

    W1 = (u2 - 1j*v2) * dz2_dz1

    return W1.real, -W1.imag

def plot_physical_geometry_R2L(blades_z1, num_blades, stagger_angle,
    alpha_deg):
    plt.figure(figsize=(8, 10))
    plt.title(f"Physical Plane Geometry ($z_1$)\nFlow Right $\\to$ Left
    ")
    for i, blade in enumerate(blades_z1):
        color = 'blue' if i == num_blades//2 else 'gray'
        plt.plot(blade.real, blade.imag, color=color, alpha=0.5 if i!=
    num_blades//2 else 1)
        plt.fill(blade.real, blade.imag, color=color, alpha=0.1)

    x_start = 1.0; y_start = 0.0; arrow_len = 0.5
```

```python
    vx = -np.cos(np.radians(alpha_deg))
    vy = np.sin(np.radians(alpha_deg))

    plt.arrow(x_start, y_start, vx*arrow_len, vy*arrow_len,
              head_width=0.05, fc='green', ec='green', lw=2)
    plt.text(x_start, y_start, f"  Inlet Flow\n  $\\alpha={alpha_deg}^\circ$", color='green', va='bottom')
    plt.xlabel("$x_1$"); plt.ylabel("$y_1$"); plt.axis('equal'); plt.grid(True, linestyle='--', alpha=0.6)
    plt.show()

def plot_transformed_geometry_R2L(blades_z2, num_blades):
    plt.figure(figsize=(10, 6))
    plt.title(f"Transformed Plane Geometry ($z_2$)\nInlet at (+1), Outlet at (-1)")
    for i, blade_z2 in enumerate(blades_z2):
        color = 'blue' if i == num_blades//2 else 'red'
        plt.plot(blade_z2.real, blade_z2.imag, color=color, alpha=0.3 if i!=num_blades//2 else 1)
    plt.scatter([1], [0], color='green', s=100, label='Inlet (+1)', zorder=5)
    plt.scatter([-1], [0], color='red', s=100, label='Outlet (-1)', zorder=5)
    plt.xlabel("$x_2$"); plt.ylabel("$y_2$"); plt.axis('equal'); plt.grid(True, linestyle='--', alpha=0.6)
    plt.legend(); plt.show()

def plot_cp_curve(z2_mid, Cp_panels, spacing, alpha_deg, stagger_deg):
    z1_mid = (spacing / np.pi) * np.arctanh(z2_mid)
    x_phys = z1_mid.real
    x_max = np.max(x_phys); x_min = np.min(x_phys)
    x_plot = (x_max - x_phys) / (x_max - x_min)

    plt.figure(figsize=(10, 6))
    trim = 2
    plt.plot(x_plot[trim:-trim], Cp_panels[trim:-trim], 'b-o', markersize=3,
             label=f'$\\alpha={alpha_deg}^\circ$')

    plt.xlabel('x/c (0 = Leading Edge)'); plt.ylabel('Coefficient of Pressure ($C_p$)')
    plt.title(f'Pressure Distribution (Positive Up)\nStagger={stagger_deg}$^\circ$, Alpha={alpha_deg}$^\circ$')
    plt.grid(True, linestyle='--', which='both'); plt.legend(); plt.show()

def plot_physical_flow(X_grid, Y_grid, U_grid, V_grid, blades_z1, stagger_angle, alpha_deg):
    plt.figure(figsize=(10, 8))
    plt.title(f"Physical Plane Flow Field ($z_1$)\nStagger $\\beta={stagger_angle}^\circ$, AoA $\\alpha={alpha_deg}^\circ$")

    U_plot = U_grid.copy()
    V_plot = V_grid.copy()

    points_flat = np.column_stack((X_grid.flatten(), Y_grid.flatten()))

    combined_mask = np.zeros(X_grid.size, dtype=bool)
```

```python

    for blade in blades_z1:
        blade_poly = np.column_stack((blade.real, blade.imag))
        path = Path(blade_poly)
        is_inside = path.contains_points(points_flat)
        combined_mask = np.logical_or(combined_mask, is_inside)

    mask_grid = combined_mask.reshape(X_grid.shape)
    U_plot[mask_grid] = np.nan
    V_plot[mask_grid] = np.nan

    plt.streamplot(X_grid, Y_grid, U_plot, V_plot, color='cyan',
   density=1.5, arrowsize=1.5)

    for i, blade in enumerate(blades_z1):
        plt.fill(blade.real, blade.imag, 'gray', alpha=0.3, zorder=3)
        plt.plot(blade.real, blade.imag, 'k', linewidth=1.5, zorder=3)

    plt.xlabel("$x_1$")
    plt.ylabel("$y_1$")
    plt.axis('equal')
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.show()

def plot_transformed_flow_R2L(gammas, Gamma_down, Q, Gamma_up,
   blades_z2_nodes, alpha_deg):
    grid_res = 200
    y_grid, x_grid = np.mgrid[-1.5:1.5:200j, -1.5:1.5:200j]
    z2_grid = x_grid + 1j * y_grid

    u2 = np.zeros_like(x_grid)
    v2 = np.zeros_like(y_grid)

    XB = blades_z2_nodes.real
    YB = blades_z2_nodes.imag
    X_panel = 0.5 * (XB[:-1] + XB[1:])
    Y_panel = 0.5 * (YB[:-1] + YB[1:])
    dx = XB[1:] - XB[:-1]; dy = YB[1:] - YB[:-1]
    S = np.sqrt(dx**2 + dy**2)

    for k in range(len(gammas)-1):
        rx = z2_grid.real - X_panel[k]
        ry = z2_grid.imag - Y_panel[k]
        r2 = rx**2 + ry**2

        circ_k = gammas[k] * S[k]
        u2 += -(1.0 / (2*np.pi)) * ry / r2 * circ_k
        v2 +=  (1.0 / (2*np.pi)) * rx / r2 * circ_k

    def add_sing(x0, y0, str_val, is_src):
        rx = z2_grid.real - x0
        ry = z2_grid.imag - y0
        r2 = rx**2 + ry**2 + 1e-9
        if is_src:
            u2[:] += (str_val / (2*np.pi)) * rx / r2
            v2[:] += (str_val / (2*np.pi)) * ry / r2
        else:
            u2[:] += -(str_val / (2*np.pi)) * ry / r2
```

```
            v2[:] +=  (str_val / (2*np.pi)) * rx / r2

    add_sing(1.0, 0.0, Q, is_src=True)
    add_sing(1.0, 0.0, Gamma_up, is_src=False)
    add_sing(-1.0, 0.0, -Q, is_src=True)
    add_sing(-1.0, 0.0, Gamma_down, is_src=False)

    blade_polygon = np.column_stack((XB, YB))
    path = Path(blade_polygon)

    points_flat = np.column_stack((x_grid.flatten(), y_grid.flatten()))
    is_inside = path.contains_points(points_flat)

    mask = is_inside.reshape(x_grid.shape)

    u2[mask] = np.nan
    v2[mask] = np.nan

    plt.figure(figsize=(10, 6))
    plt.title(f"Transformed Plane ($z_2$) Flow Field\nInlet(+1) $\\to$
   Outlet(-1) ($\\alpha={alpha_deg}^\circ$)")

    plt.streamplot(x_grid, y_grid, u2, v2, color='orange', density=1.5,
    arrowsize=1.5)

    plt.plot(XB, YB, 'k-', linewidth=2.5, label='Transformed Blade')
    plt.fill(XB, YB, 'gray', alpha=0.3)

    plt.scatter([1], [0], color='green', s=100, label='Inlet (+1)',
   zorder=5)
    plt.scatter([-1], [0], color='red', s=100, label='Outlet (-1)',
   zorder=5)

    plt.xlabel("$x_2$")
    plt.ylabel("$y_2$")
    plt.axis('equal')
    plt.grid(True, linestyle='--', alpha=0.6)
    plt.legend(loc='upper center')
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    naca_code = '0012'
    chord = 1.0;
    spacing = 1.0;
    num_blades = 3;
    N_panels = 200

    alpha_deg = -10.0
    stagger_angle = 20.0
    V_inlet = 10.0

    x_base, y_base = get_naca_4_digit_airfoil(naca_code, c=chord,
   n_points=N_panels//2 + 1)
    x_base = -(x_base - 0.5)
    x_base = x_base[::-1]; y_base = y_base[::-1]
    x_rot, y_rot = rotate_coords(x_base, y_base, stagger_angle)

```

```
    blades_z1 = []
    center_idx = num_blades // 2
    for i in range(num_blades):
        shift = i - center_idx
        blades_z1.append((x_rot + 1j * y_rot) + (1j * shift * spacing))

    z1_ref = blades_z1[center_idx]
    z2_ref = np.tanh(np.pi * z1_ref / spacing)
    blades_z2 = [np.tanh(np.pi * b / spacing) for b in blades_z1]

    Cp, z2_mid, gammas, G_down, Q, G_up =
solve_cascade_panel_method_R2L(z2_ref, V_inlet, alpha_deg, spacing,
chord)

    plot_physical_geometry_R2L(blades_z1, num_blades, stagger_angle,
alpha_deg)
    plot_transformed_geometry_R2L(blades_z2, num_blades)
    plot_cp_curve(z2_mid, Cp, spacing, alpha_deg, stagger_angle)

    print("  > Calculating Flow Field Grids...")
    grid_res = 100
    x_f = np.linspace(-1.5, 1.5, grid_res); y_f = np.linspace(-2.0,
2.0, grid_res)
    X_grid, Y_grid = np.meshgrid(x_f, y_f)

    U_grid, V_grid = compute_flow_field(X_grid, Y_grid, gammas, G_down,
    Q, G_up, z2_ref, spacing)

    u_inlet = U_grid[50, -1]
    v_inlet = V_grid[50, -1]
    calc_alpha = np.degrees(np.arctan2(v_inlet, -u_inlet))
    print(f"  > Verification: Calculated Inlet Angle = {calc_alpha:.2f}
    degrees")

    plot_physical_flow(X_grid, Y_grid, U_grid, V_grid, blades_z1,
stagger_angle, alpha_deg)
    plot_transformed_flow_R2L(gammas, G_down, Q, G_up, z2_ref,
alpha_deg)
```

Listing 1: Python script for Cascade Potential Flow using Conformal Mapping.

5 Results and Discussion

The Python code was executed for a cascade of NACA 0012 airfoils with spacing $h = 1.0$, chord $c = 1.0$, stagger angle $\beta = 20°$ and angle of attack $\alpha = -10°$.

5.1 Physical and Transformed Geometry

Figure 2 illustrates the transformation. The linear cascade of airfoils (left) maps to a single closed loop in the computational plane (right). The upstream infinity maps to $z_2 = 1$ (Green dot), and downstream to $z_2 = -1$ (Red dot).

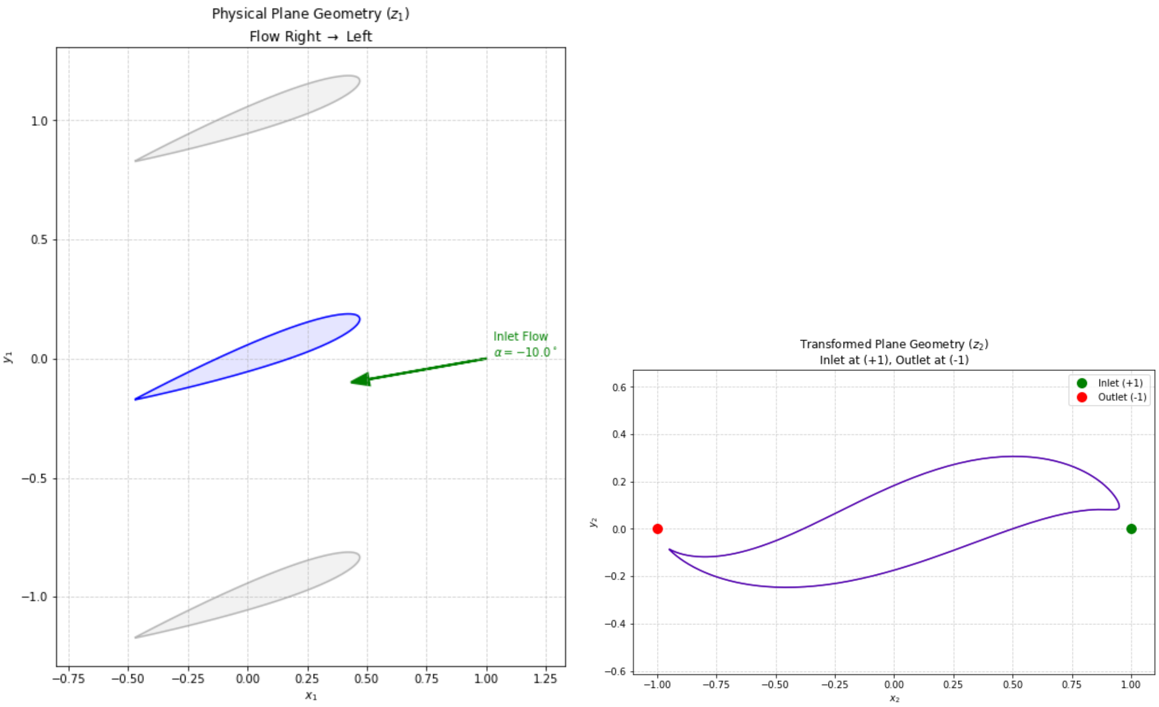

Figure 2: Left: Physical cascade (z_1). Right: Transformed single body (z_2).

5.2 Pressure Coefficient Distribution

Figure 3 shows the C_p distribution over the blade surface. Unlike the isolated airfoil case, the cascade effect alters the effective angle of attack and local velocity magnitudes due to blade-to-blade interference.

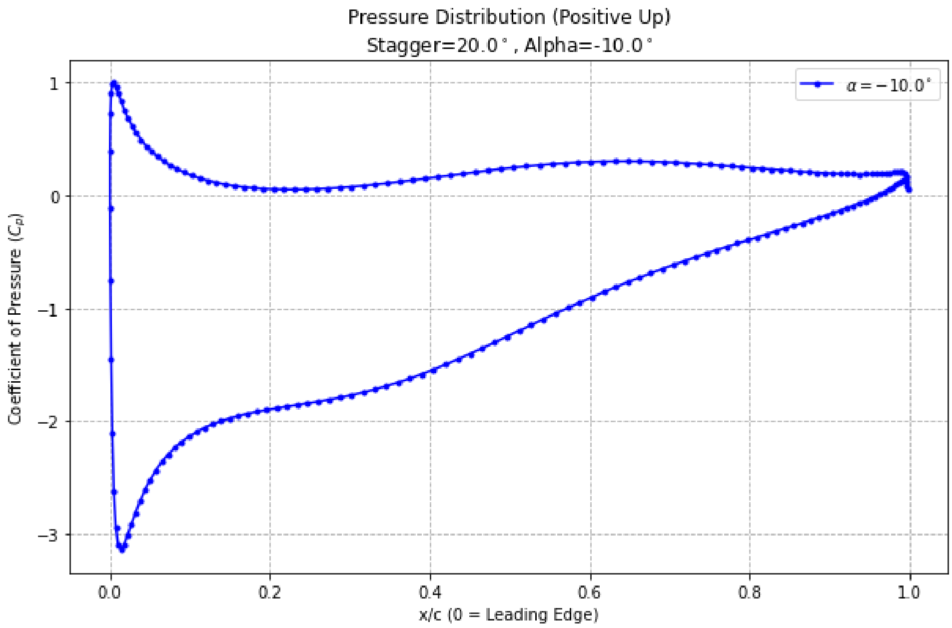

Figure 3: Pressure Coefficient distribution for the central blade.

5.3 Flow Field Visualization

The streamlines are visualized in both planes. In the z_2-plane, flow moves from the source at (+1) to the sink at (-1) around the obstacle. When mapped back to z_1, this corresponds to the flow turning through the cascade.

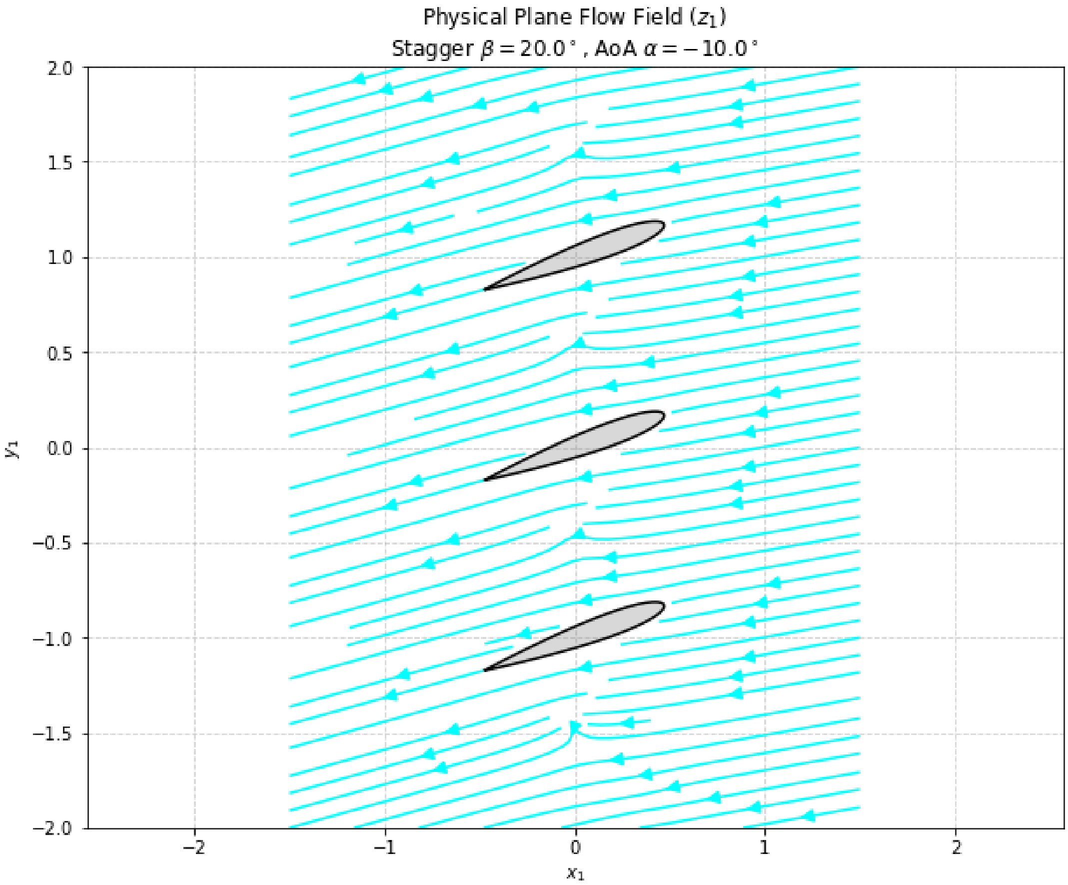

Figure 4: Computed streamlines through the cascade.

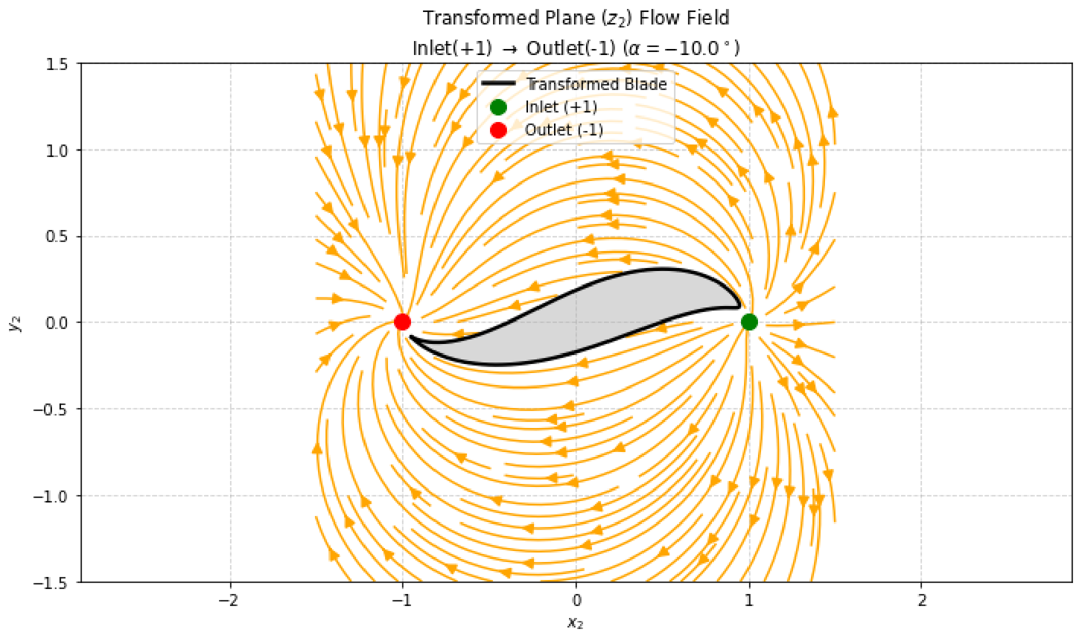

Figure 5: Computed streamlines in Z2 plane.

6 References

1. Bhimarasetty, A., & Govardhan, R. N. (2010). A simple method for potential flow simulation of cascades. *Sadhana*, 35(6), 649-657.

2. Kuethe, A. M., & Chow, C. Y. (1986). *Foundations of Aerodynamics*. Wiley.