Tugas AE 6000 Mekanika Kontinum II
Wing Simulation using Panel Method (Source and Doublet)

Arranged by:

Dhanika	23620006
Edwin Aldrian Santoso	23620021
Muhammad Faiz Izzaturrahman	23620029

Graduate Program in Aerospace Engineering
Faculty of Mechanical and Aerospace Engineering
Institut Teknologi Bandung

2020

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Background

Analysis and design of airfoil are one of the most important processes in the design of aerospace products. Because of cost and time reason, a simple computational method is developed, Viscous-Inviscid Interaction (VII). VII method is a method that divides the fluid domain into two parts, that is viscous domain and inviscid domain. Solutions of the inviscid domain outside of the viscous domain are computed by using Laplace equation and solutions of the viscous domain are computed by using Boundary Layer equations.

Inviscid flow with assumptions: incompressible irrotational, and steady-state satisfy Laplace equation. Because the nature of Laplace equation is elliptic differential equations, then the solution depends on boundary conditions of the domain. There are many methods can be used to solve this equation and in this project, authors use Panel Method which is one of Boundary Element Method (BEM). Panel Method using linear combination of solutions from singularity element. Boundary conditions for flow around the wing are freestream and there is no flow pass through the wing. In this project, authors developed the Python source code to solve flow around the wing using Panel Method.

1.2 Goals

Goals of this project are:

1. Make source code to solve flow around the wing using Panel Method.
2. Study effects of number of panels to the results.
3. Study effects of angle of wake to the results.

Chapter 2 Theoretical Background

2.1 The General Solution to the Potential Flow Problem

Given an incompressible, irrotational fluid, the continuity equation is defined by the Laplace equation,

$$\nabla^2 \phi = 0 \tag{2.1}$$

Whereby, a solid submerged in the fluid is subject to a Neumann boundary condition. Furthermore, with $\nabla\phi$ being measured in a frame of reference attached to the body, the disturbance due to the motion would decay far from the body.

$$\begin{aligned} \nabla\phi \cdot \mathbf{n} &= 0 \\ \lim_{r\to\infty} (\nabla\phi - \mathbf{v}) &= 0 \end{aligned} \tag{2.2}$$

Where $\mathbf{v}$ is the relative velocity, at infinity, seen by the observer moving with the body.

For an arbitrary body with boundary S_B within a volume V bounded by an outer S_∞, the boundary conditions of Eq. 2.2 apply to S_B and S_∞ of Figure 2.1

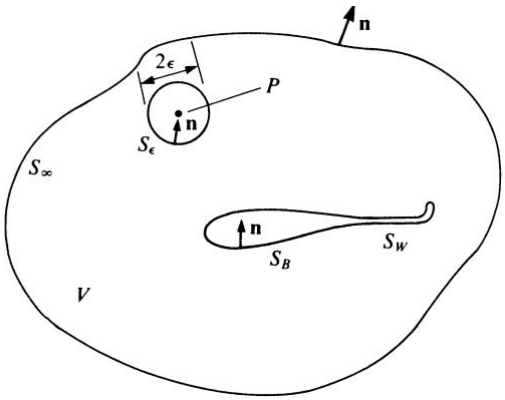


Figure 2.1: The potential flow problem

respectively. In order to solve Eq. 2.1 we introduce Green's second identity for two potential scalar functions of position, namely, Φ_1 and Φ_2 over all boundaries $S = S_B + S_W + S_\infty$. Where S_W is the prescribed boundary for the wake.

$$\int_S (\Phi_1 \nabla \Phi_2 - \Phi_2 \nabla \Phi_1) \cdot \mathbf{n} \; dS = \int_S (\Phi_1 \nabla^2 \Phi_2 - \Phi_2 \nabla^2 \Phi_1) \; dV \tag{2.3}$$

Consider the situation if Figure 2.1, where a point of interest P is within the region of V. If we define,

$$\Phi_1 = \frac{1}{r} \; and \; \Phi_2 = \Phi \tag{2.4}$$

where r is the distance from the point P. When P is within the region V it must be excluded from the region of integration. Thus, we define a small sphere of radius ϵ that encompasses point P. We see that outside the sphere, with the definition of Eq. 2.4, as $r \to \infty$ the Laplace equation is satisfied. Rewriting Eq. 2.3 in a spherical coordinate system at point P

$$-\int_{sphere \; \epsilon} \left(\frac{1}{r} \frac{\partial \Phi}{\partial r} + \frac{\Phi}{r^2} \right) \; dS + \int_S \left(\frac{1}{r} \nabla \Phi - \Phi \nabla \frac{1}{r} \right) \cdot \mathbf{n} \; dS = 0 \tag{2.5}$$

Where, as of Figure 2.1, $\mathbf{n} = -\mathbf{e}_r$, $\mathbf{n} \cdot \nabla \Phi = -\frac{\partial \Phi}{\partial r}$ and $\nabla \frac{1}{r} = (\frac{1}{r^2} \mathbf{e}_r)$.

By taking $r = \epsilon$ and $\epsilon \to 0$ we obtain a formula for the potential Φ at any point in the flow in region V. Eq. 2.5 becomes,

$$\Phi(P) = \frac{1}{4\pi} \int_S \left(\frac{1}{r} \nabla \Phi - \Phi \nabla \frac{1}{r} \right) \cdot \mathbf{n} \; dS \tag{2.6}$$

With Eq. 2.6 we can derive several formulations for the potential Φ at any point P on any surface S in terms of the values of Φ and $\frac{\partial \Phi}{\partial n}$. Of particular interest, is the potential defined in the boundary S_B. Assuming that a body is moving through a stationary fluid (Φ_∞ is selected as a constant in the region V) and the wake surface, S_W is thin such that $\frac{\partial \Phi}{\partial n}$ is continuous, then $\Phi(P)$ becomes,

$$\Phi(P) = \frac{1}{4\pi} \int_{S_B} \left(\frac{1}{r} \nabla (\Phi - \Phi_i) - (\Phi - \Phi_i) \nabla \frac{1}{r} \right) \cdot \mathbf{n} \; dS - \frac{1}{4\pi} \int_{S_W} \Phi \mathbf{n} \cdot \nabla \frac{1}{r} \; dS + \Phi_\infty(P) \tag{2.7}$$

Where Φ_i is defined as the *inner potential* inside the region of S_B, the description of the problem of Eq. 2.7 is visualized in Figure 2.2. Therefore, the problem Eq. 2.6 reduces to finding the values for Φ and $\frac{\partial \Phi}{\partial n}$, rather, the difference in the former and latter between their inner and outer values. The said *difference* is

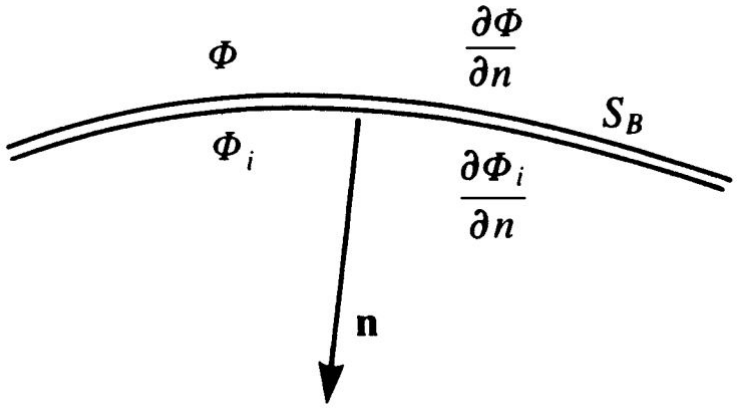


Figure 2.2: Velocity potential Φ near a solid boundary S_B

what we call the *elementary solutions*. Namely, the *doublet* μ for the difference in Φ and *source* σ for the difference in $\frac{\partial \Phi}{\partial n}$. Such that

$$-\mu = \Phi - \Phi_i \; and - \sigma = \frac{\partial \Phi}{\partial n} - \frac{\partial \Phi_i}{\partial n} \tag{2.8}$$

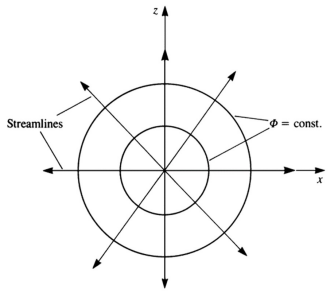
2.2 Singularity Elements

Essentially by distributing the elementary solutions over the boundaries S_B and S_W, as described in the previous section, the solution of Eq. 2.1 can be obtained. These elementary solutions automatically fulfill the boundary conditions of Eq. 2.2 as $r \to \infty$. However, when $r = 0$ we obtain a velocity that reaches to infinity. These solutions are called *singular solutions* and the general solution to Eq. 2.1 requires the integration of these solutions over any surface S. Therefore, the problem is now reduced to finding the most appropriate singularity element distribution over the known boundaries. Using Eq. 2.8, Eq. 2.7 becomes,

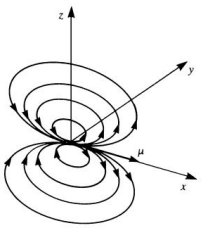
$$\Phi(P) = -\frac{1}{4\pi}\int_{S_B}\left[\sigma\left(\frac{1}{r}\right) - \mu\mathbf{n}\cdot\nabla\left(\frac{1}{r}\right)\right] dS + \frac{1}{4\pi}\int_{S_W}\left[\mu\mathbf{n}\cdot\nabla\left(\frac{1}{r}\right)\right] dS + \Phi_\infty(P) \tag{2.9}$$

2.2.1 Source

A basic solution to Eq. 2.9 is the point source/sink. If a point element is located at some $\mathbf{r_0}$, then the corresponding potential and velocity, expressed in Cartesian coordinates (x, y, z), becomes,



(a) Source/Sink streamlines and equipotential lines at the origin



(b) Doublet Streamlines

$$\Phi(x,y,z) = \frac{-\sigma}{4\pi\sqrt{(x-x_0)^2+(y-y_0)^2+(z-z_0)^2}} \tag{2.10}$$

$$\mathbf{u} = \left(\frac{\partial\Phi}{\partial x}, \frac{\partial\Phi}{\partial y}, \frac{\partial\Phi}{\partial z}\right) \tag{2.11}$$

2.2.2 Doublet

Another solution to Eq. 2.9 is the point doublet. In Cartesian coordinates, the potential becomes,

$$\Phi(x,y,z) = \frac{\mu}{4\pi}\begin{bmatrix}\frac{\partial}{\partial x}\\ \frac{\partial}{\partial z}\\ \frac{\partial}{\partial z}\end{bmatrix}\frac{1}{\sqrt{(x-x_0)^2+(y-y_0)^2+(z-z_0)^2}} \tag{2.12}$$

Similar to Eq. 2.11, to get the velocity we differentiate Eq. 2.12.

2.3 Boundary Conditions

Recall Eq. 2.9,

$$\Phi^*(P) = -\frac{1}{4\pi}\int_{S_B}\left[\sigma\left(\frac{1}{r}\right) - \mu\mathbf{n}\cdot\nabla\left(\frac{1}{r}\right)\right]dS + \frac{1}{4\pi}\int_{S_W}\left[\mu\mathbf{n}\cdot\nabla\left(\frac{1}{r}\right)\right]dS + \Phi_\infty(P) \tag{2.13}$$

For three-dimensional flow specifying the boundary conditions will not immediately output a unique solution due to two problems. First being the right decision of the combination of source and doublet distributions need to be chosen and second an extra physical condition needs to be enforced to fix the amount of

circulation around the surface S_B. This concerns the modelling of the wakes and fixing the wake shedding lines. Therefore, we rewrite Eq 2.13 as,

$$\Phi^*(P) = -\frac{1}{4\pi}\int_{body}\left[\sigma\left(\frac{1}{r}\right)\right]dS + \frac{1}{4\pi}\int_{body+wake}\left[\mu\mathbf{n}\cdot\nabla\left(\frac{1}{r}\right)\right]\ dS + \Phi_\infty(P) \tag{2.14}$$

2.3.1 Neumann

If we were to specify $\frac{\partial\Phi^*}{\partial n}$ across S_B,

$$\nabla(\Phi + \Phi_\infty)\cdot\mathbf{n} = 0 \tag{2.15}$$

where Φ is the perturbation potential. To satisfy Eq. 2.15 we use the velocity field due to the singularity distribution.

$$\nabla\Phi^*(P) = -\frac{1}{4\pi}\int_{body}\left[\sigma\nabla\left(\frac{1}{r}\right)\right]dS + \frac{1}{4\pi}\int_{body+wake}\left[\mu\nabla\left(\frac{\partial}{\partial n}\frac{1}{r}\right)\right]\ dS + \Phi_\infty(P) \tag{2.16}$$

Substituting into Eq. 2.15,

$$\Big\{-\frac{1}{4\pi}\int_{body}\left[\sigma\nabla\left(\frac{1}{r}\right)\right]dS + \frac{1}{4\pi}\int_{body+wake}\left[\mu\nabla\left(\frac{\partial}{\partial n}\frac{1}{r}\right)\right]\ dS + \Phi_\infty(P)\Big\}\cdot n = 0 \tag{2.17}$$

For numerical solutions, Eq. 2.17 should hold, on every collocation point in S_B. When specified on these points in terms of the unknown singularities then Eq. **??** reduce to a set of algebraic equations.

2.3.2 Dirichlet

If we were to specify Φ^* at the boundary S_B, we would have the Dirichlet problem. By distributing the singularity elements across the surface and placing point P inside S_B, we can obtain the inner potential Φ^* as,

$$\Phi_i^*(P) = -\frac{1}{4\pi}\int_{body}\left[\sigma\left(\frac{1}{r}\right)\right]dS + \frac{1}{4\pi}\int_{body+wake}\left[\mu\frac{\partial}{\partial n}\left(\frac{1}{r}\right)\right]\ dS + \Phi_\infty(P) \tag{2.18}$$

For the Dirichlet boundary condition we have,

$$\Phi_i^* = (\Phi + \Phi_\infty)_i = const \tag{2.19}$$

Or in other words Eq. 2.18 is equal to a constant. if we set Eq. 2.19 to Φ_∞, Eq 2.18 simplifies to,

$$\frac{1}{4\pi}\int_{body+wake}\mu\frac{\partial}{\partial n}\left(\frac{1}{r}\right)\ dS - \frac{1}{4\pi}\int_{body}\sigma\left(\frac{1}{r}\right)\ dS = 0 \tag{2.20}$$

Consequently, Eq. 2.20 is based on the fact that if the Neumann boundary condition ($\frac{\partial \Phi^*}{\partial n} = 0$) is equivalent to,

$$\frac{\partial \Phi}{\partial n} = -\mathbf{n} \cdot \mathbf{Q}_\infty \tag{2.21}$$

and that σ is defined as Eq. 2.8. Since Φ_i and its normal derivative is zero on S_B then in order for Eq. 2.20 to be true, the source strength is required to be,

$$\sigma = \mathbf{n} \cdot \mathbf{Q}_\infty \tag{2.22}$$

2.4 Wake Model

To address the second problem as discussed in the previous section, after selecting a good combination of sources and doublets with the boundary conditions fulfilled on S_B, we specify two additional conditions to obtain a unique solution. That is for a wake model,

- The wake strength at the trailing edge needs to be set
- The wake shape and location needs to be set.

Concerning the wake strength, the simplest solution is to apply the two-dimensional Kutta condition along the wing trailing edge, that is

$$\gamma_{TE} = 0 \tag{2.23}$$

Rewriting the above, in terms of the doublet strengths we get,

$$\mu_{uppersurface} - \mu_{lowersurface} - \mu_{wake} = 0 \tag{2.24}$$

Similarly, in terms of the circulation to define the strength of a wake panel,

$$\Gamma_{wake} = \Gamma_{uppersurface} - \Gamma_{lowersurface} \tag{2.25}$$

On the wake shape in three dimensions, the wake influence is more dominant and thus its geometry affects the solution. Recall the formulation for a force $\Delta\mathbf{F}$ generated by a vortex sheet γ.

$$\Delta\mathbf{F} = \rho\mathbf{q} \times \gamma \tag{2.26}$$

For the three dimensional case $\delta\mathbf{F} = 0$ when the local flow is parallel to γ. Therefore, the condition for the wake geometry is

$$\mathbf{q} \times \gamma_{wake} = 0 \tag{2.27}$$

Similarly, for a wake represented by a thin doublet sheet,

$$\mathbf{q} \times \mu_{wake} = 0 \tag{2.28}$$

Thus, the condition required for the wake panels, in terms of doublets is

$$\mu_{wake} = constant \tag{2.29}$$

and the elements boundaries are required to be parallel to the local streamlines. The difficulty in satisfying this condition is that the wake location is not known apriori.

Chapter 3 Numerical Algorithm

3.1 Flowchart

The numerical algorithm discussed in this chapter is the way how authors made the source code to simulate the inviscid flow around wing using Panel method. The summary of the code making process was presented in flow chart as can be seen in Figure 3.1.

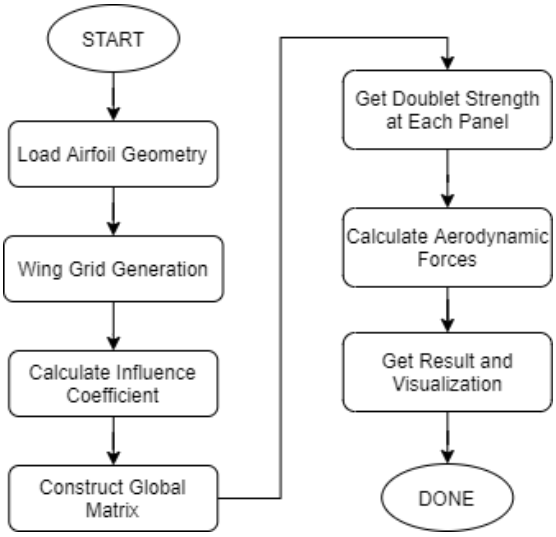


Figure 3.1: Flowchart Diagram

The first step is to load the airfoil geometry profile, in our program we extracted the data from folder which contains many airfoil coordinate files. We also added some algorithm to increase or decrease the number of airfoil points in our program using cubic spline interpolation, such that the number of airfoil points was not limited to the number of points in the airfoil coordinates file but can be adjusted according to the needs.

The second step is creating the 3D wing model by expanding the cross section (airfoil) profile in span-wise direction. Some parameters need to be determined to

ensure the wing geometry such as wing span, root chord, tip chord, and swept angle (aft sweep of tip). Another important parameters are the data for free-stream velocity and wing angle of attack. At the same time, the geometry definition was implemented by creating grid at the wing surface. The discretization is controlled by determining the number of span-wise panels and airfoil coordinate points. Additional grid needs to be made which is the wake. The simulation grid was only half-wing configuration to make the computation process more efficient.

The next step is calculating the strength of source and doublet at each panel. Calculating the velocity from the integral form of source and doublet in this problem was quite complicated and because the summation should be done also from each section of panels using its own local coordinates, so the implementation was divided into two parts. First, local frame of reference at each panel was calculated, then it this result was used to calculate the influence parameter of source and doublet. The velocity at each panel could be calculated from the summation of the influence parameters. To fulfill the boundary condition which is normal velocity at wing panel should be zero, all of these boundary condition equations were constructed to global matrix to solve the doublet strength. Additional boundary condition is added to the global matrix which is the Kutta condition at the trailing edge of the wing to ensure the flow at trailing edge parallel to the airfoil and the solution is physical.

The last step is to calculate the aerodynamic parameters which are pressure coefficient (C_p), lift coefficient (C_l), and moment coefficient (C_m). The drag coefficient is not resolved well in this inviscid simulation because the viscous effect is neglected, additional inviscid-viscous interaction simulation could be added to resolve the drag coefficient. Additional visualization is needed to understand the result better. In our program, the visualization was made using Matplotlib library in Python and ParaView.

3.2 Grid Generation

The grid was made on the wing surface as panels and additional wake panels at the trailing edge. The global Cartesian coordinate is chosen as follows: x axis is the chord-wise direction, y axis is the span-wise direction, and z axis is the direction from the cross product between x axis and y axis. In the computation domain, the grid points were stored in array containing (x, y, and z) values. There were several types of grid points which were calculated for each panel.

The first type is the corner point at each panel. So, each panel was constructed by four corner points, then the second type is the collocation points at each panel. This type of point is located at the center of each panel and calculated from averaging from four corner points at corresponding panel. The visualization of the grid generation containing panels and collocations points was shown in Figure 3.2.

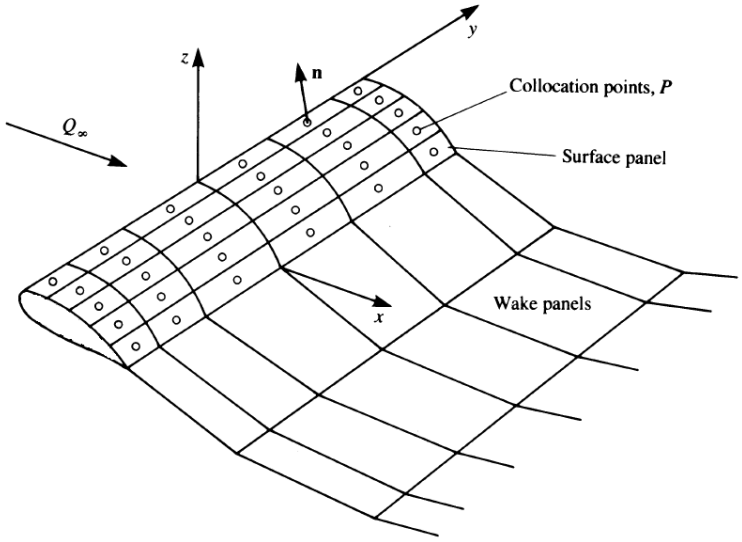


Figure 3.2: Grid Panels Along Wing Surface and Wake Panels

3.2.1 Wake

Wake at the wing trailing edge needs to be introduced in this three-dimensional simulation since the bound vorticity needs to be continued beyond the wing and to ensure that the wing to have circulation (Γ) at a span-wise location, so a discontinuity in the velocity potential near the trailing edge must exist.

$$\Phi_2 - \Phi_1 = \Gamma \tag{3.1}$$

In our simulation, the wake was created by adding only one long panel at the wing trailing edge whose angle can vary. The length of wake panel was set about 100 times the wing span. The strength of doublet at the wake was calculated from the boundary condition at the trailing edge (Kutta-Condition).

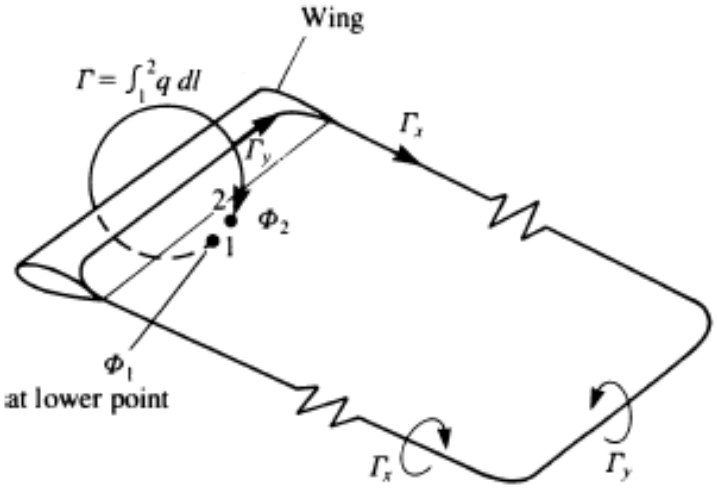


Figure 3.3: Wing Vorticity

3.3 Aerodynamic Calculation

This section will discuss about the process of obtaining the strength of source (σ) and doublet (μ) at each panel such that the numerical approximation meets the physical boundary condition requirements. Dirichlet boundary condition was chosen for this simulation, it requires that the potential velocity inside the body (Φ_i^*) should be constant. The expression of summation of the effect from source and doublet follows Eq 2.18 and can be simplified to Eq 2.20.

The integration in Eq 2.20 becomes more convenient if we calculate them in local frame of reference at each panel. It will be used then in terms of influence during the global matrix construction process. At each panel, the center of the coordinate is the collocation point, the coordinate used also a local coordinate which are chord-wise, span-wise, and normal, unique for each panel.

3.3.1 Influence

The integration result in Eq 2.20 for source and doublet from each panel to specific coordinate can be calculated as the influence coefficient. First, the source strength is defined by Eq 2.22 to fulfill the boundary condition, the remaining task is to find the doublet strength distribution. The influence coefficient both for source and doublet are needed to construct the global matrix then find the doublet strength distribution.

The influence coefficient was calculated using quadrilateral constant source or

doublet to a certain point which is the target panel collocation point as can be seen in Figure 3.4 the influence from panel k to another panel. This procedure was repeated for each pair of panels to get all of the influence coefficients.

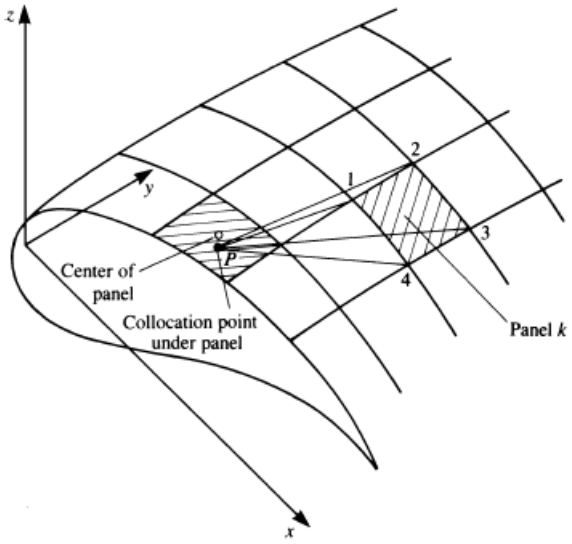


Figure 3.4: Influence Schematic From Panel k

For quadrilateral constant strength source (σ), the integral form of velocity potential was given in Eq 3.2.The x_o and y_o is the center of local coordinate (collocation point).

$$\Phi_s(x,y,z) = \frac{-\sigma}{4\pi}\int_S \frac{dS}{\sqrt{(x-x_o)^2+(y-y_o)^2}+z^2} \tag{3.2}$$

In local coordinate the equation in summation form was given in Eq 3.4 based on the coordinate of corner points of the panel with index (1,2,3,and 4) as shown in Figure 3.4. This is the convention of index numbering that was used in our simulation code. The x, y, and z axis are local coordinate axis at each panel. Some additional parameters in Eq 3.3 were defined to construct Eq 3.4.

$$\begin{aligned} d_{ij} &= \sqrt{(x_j-x_i)^2+(y_j-y_i)^2} \\ m_{ij} &= \frac{y_j-y_i}{x_j-x_i} \\ r_k &= \sqrt{(x-x_k)^2+(y-y_k)^2+z^2} \\ e_k &= (x-x_k)^2+z^2 \\ h_k &= (x-x_k)(y-y_k) \end{aligned} \tag{3.3}$$

$$\Phi_s(x,y,z) = \frac{-\sigma}{4\pi}\Bigg\{\Bigg[\frac{(x-x_1)(y_2-y_1)-(y-y_1)(x_2-x_1)}{d_{12}} ln\left(\frac{r_1+r_2+d_{12}}{r_1+r_2-d_{12}}\right) + \frac{(x-x_2)(y_3-y_2)-(y-y_2)(x_3-x_2)}{d_{23}} ln\left(\frac{r_2+r_3+d_{23}}{r_2+r_3-d_{23}}\right) + \frac{(x-x_3)(y_4-y_3)-(y-y_3)(x_4-x_3)}{d_{34}} ln\left(\frac{r_3+r_4+d_{34}}{r_3+r_4-d_{34}}\right) + \frac{(x-x_4)(y_1-y_4)-(y-y_4)(x_1-x_4)}{d_{41}} ln\left(\frac{r_4+r_1+d_{41}}{r_4+r_1-d_{41}}\right)\Bigg] - |z|\Bigg[tan^{-1}\left(\frac{m_{12}e_1-h_1}{zr_1}\right) - tan^{-1}\left(\frac{m_{12}e_2-h_2}{zr_2}\right) + tan^{-1}\left(\frac{m_{23}e_2-h_2}{zr_2}\right) - tan^{-1}\left(\frac{m_{23}e_3-h_3}{zr_3}\right) + tan^{-1}\left(\frac{m_{34}e_3-h_3}{zr_3}\right) - tan^{-1}\left(\frac{m_{34}e_4-h_4}{zr_4}\right) + tan^{-1}\left(\frac{m_{41}e_4-h_4}{zr_4}\right) - tan^{-1}\left(\frac{m_{41}e_1-h_1}{zr_1}\right)\Bigg]\Bigg\} \quad (3.4)$$

All of the elements in Eq 3.4 except the source strength (σ) is the influence coefficient from constant source (B_k) in quadrilateral panel to some certain point (x, y, z). In our simulation, that point will be the collocation point of other panel.

For quadrilateral constant strength doublet (μ), the integral form was given in Eq 3.5 and the summation form in local coordinate was given in Eq 3.6. The doublet influence coefficient (C_k) is all of the elements in Eq 3.6 except the doublet strength.

$$\Phi_d(x,y,z) = \frac{-\mu}{4\pi}\int_S \frac{z dS}{\left[(x-x_o)^2+(y-y_o)^2+z^2\right]^{3/2}} \quad (3.5)$$

$$\Phi_d(x,y,z) = \frac{\mu}{4\pi}\Bigg[tan^{-1}\left(\frac{m_{12}e_1-h_1}{zr_1}\right) - tan^{-1}\left(\frac{m_{12}e_2-h_2}{zr_2}\right) + tan^{-1}\left(\frac{m_{23}e_1-h_2}{zr_2}\right) - tan^{-1}\left(\frac{m_{23}e_3-h_2}{zr_3}\right) + tan^{-1}\left(\frac{m_{34}e_3-h_3}{zr_3}\right) - tan^{-1}\left(\frac{m_{34}e_4-h_4}{zr_4}\right) + tan^{-1}\left(\frac{m_{41}e_4-h_4}{zr_4}\right) - tan^{-1}\left(\frac{m_{41}e_1-h_1}{zr_1}\right)\Bigg] \quad (3.6)$$

The simulation was just half wing span, so during calculating the influence coefficient the contribution from the image panel should be added. The calcu-

lation was simple, just change the y-coordinate value into negative value for the image panel.

3.3.2 Trailing Edge Condition

After calculating all the influence coefficients for both source (B_k) and doublet (C_k), we should construct them into a global matrix in order to solve the doublet strength, recall that source strength has been able to be calculated using Eq 2.22 to fulfill the boundary condition. Another boundary condition to be considered is the condition at the wing trailing edge and the wake which has been stated in Eq. 2.25.

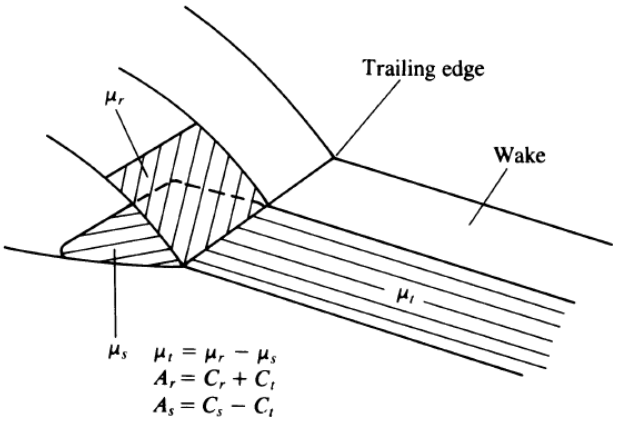


Figure 3.5: Trailing Edge Condition

If the panel is not at the trailing edge the doublet influence coefficient is still (C_k). But if the panel is at the trailing edge (T.E.) then some additional term need to be added to fulfill the trailing edge condition (Kutta-Condition). To make the calculation more convenient, parameter (A_k) was defined as substitute to (C_k) as shown in Eq 3.7. where (C_t) is the wake influence coefficient.

$$\begin{aligned} A_k &= C_k, \text{If panel is not at T.E.} \\ A_k &= C_k \pm C_t, \text{If panel is at T.E.} \end{aligned} \tag{3.7}$$

The Kutta-condition is combined inside the global matrix without adding additional row to the global matrix, it was represented by the coefficient (A_k) in panels at trailing edge.

3.3.3 Global Matrix

The global matrix construction was based on Eq 2.20 which can presented as Eq 3.9 using parameter (A_k) and (B_k) for each collocation point. The parameter of source strength has been known, so we could move this source terms to the right hand side (RHS).

$$\sum_{k=1}^{N} A_k \mu_k + \sum_{k=1}^{N} B_k \sigma_k = 0 \tag{3.8}$$

$$\sum_{k=1}^{N} A_k \mu_k = -\sum_{k=1}^{N} B_k \sigma_k \tag{3.9}$$

For all N collocation points then, it becomes matrix system with N equations to solve N unknowns from μ_1 until μ_N. The matrix system was presented in Eq 3.10.

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{pmatrix} \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_N \end{pmatrix} = - \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1N} \\ B_{21} & B_{22} & \cdots & B_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N1} & B_{N2} & \cdots & B_{NN} \end{pmatrix} \begin{pmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_N \end{pmatrix} \tag{3.10}$$

The right hand side which is the source terms is constant value and could be written as array (RHS).

$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NN} \end{pmatrix} \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_N \end{pmatrix} = \begin{pmatrix} RHS_1 \\ RHS_2 \\ \vdots \\ RHS_N \end{pmatrix} \tag{3.11}$$

The value of doublet strength (μ) distribution could be obtained by solving the matrix equation in Eq 3.11.

3.4 Aerodynamic Force

Once the doublet strength distribution was calculated by solving the global matrix, the flow information was enough and we were good to go to calculate the velocity distribution at panel using the contribution of velocity potential from free stream, source, and doublet. From the information of velocity distribution,

the pressure coefficient (C_p) at each panel could be calculated. Then, we could also compute wing lift coefficient (C_L)and moment coefficient (C_M).

In panel coordinate $(l, m.n)$, at each panel the perturbation velocity (q) could be calculated using Eq 3.12. In numerical approach, finite difference was used to calculate the derivatives.

$$\begin{aligned} q_l &= -\frac{\partial \mu}{\partial l} \\ q_m &= -\frac{\partial \mu}{\partial m} \\ q_n &= -\sigma \end{aligned} \tag{3.12}$$

The total velocity (Q) is the free stream velocity (Q_∞)plus the perturbation velocity (q).

$$\mathbf{Q} = (Q_{\infty,l} + q_l, Q_{\infty,m} + q_m, Q_{\infty,n} + q_n) \tag{3.13}$$

The pressure coefficient (C_p)at each panel (k) could be obtained using Eq 3.14.

$$C_{p,k} = 1 - \frac{Q_k^2}{Q_\infty^2} \tag{3.14}$$

The pressure force always pointed to the negative normal direction of the panels, so the pressure force should be projected in order to get lift force and drag force (in panel local coordinate). From pressure coefficient at each panel it should multiplied by panel surface area (S) to get the force contribution from that panel. Then by doing summation for all of the panels, we could obtain the lift and drag force.

Note that the simulation is just half wing, so the force calculated is just half from the actual force. It will be more convenient to provide the force calculation in form of coefficients (divide calculated lift by half span ($b/2$)).

$$C_L = \frac{\sum_{k=1}^{N} (C_{p,k} \Delta S_k r_L)}{\left(\frac{c_R + c_T}{2}\right)\left(\frac{b}{2}\right)} \tag{3.15}$$

$$C_D = \frac{\sum_{k=1}^{N} (C_{p,k} \Delta S_k r_D)}{\left(\frac{c_R + c_T}{2}\right)\left(\frac{b}{2}\right)} \tag{3.16}$$

$$C_M = \frac{\sum_{k=1}^{N} (C_{p,k} \Delta S_k r_L x_k)}{\left(\frac{c_R + c_T}{2}\right)\left(\frac{b}{2}\right) c_m} \tag{3.17}$$

where r_L and r_D are rotation factor from normal in local coordinate to normal and parallel to free stream direction respectively.c_R and c_T are root chord and root tip. In moment calculation, the lift calculation was multiplied by the moment arm (x_k) then the result was divided by mean chord (c_m).

3.5 Visualization

The visualization was conducted using ParaView using quad unstructured grid. The coordinate of edge points of each panel were printed in .vtu format which can be read by ParaView. The grid or panel visualization of zero swept wing which has airfoil NACA 0015 with 20 points and 3 panels for half wing span was shown in Figure 3.6.

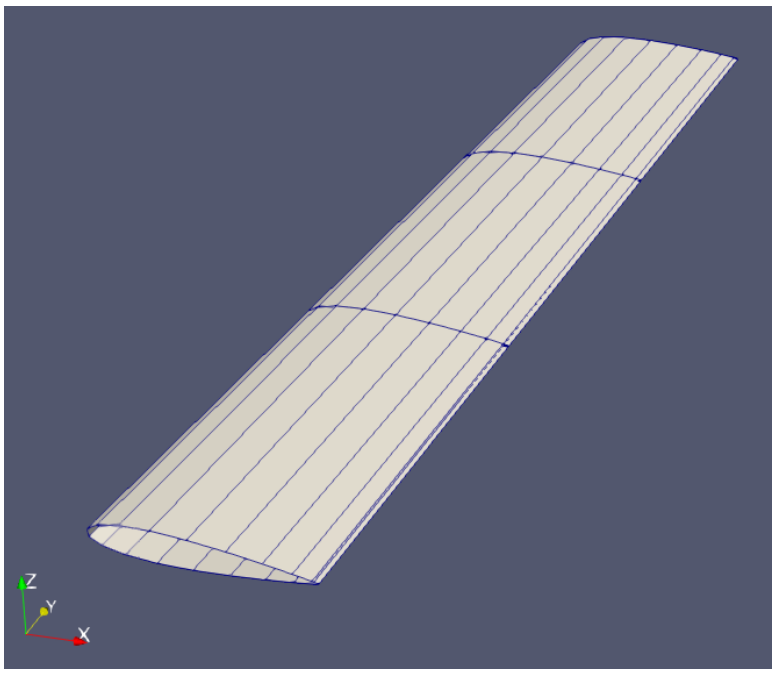


Figure 3.6: Grid Visualization using ParaView

The color map for pressure coefficient (C_p) at each panel also can be visualized using ParaView. This type of visualization will be presented as the simulation result in the Chapter 4. Result using 2D graphic data will be visualized using matplotlib library in Python.

Chapter 4 Result and Analysis

4.1 Number of Panels Variation

The simulation will be ran using several number of panels combination. We will start will small number of panels then start to increase the panel numbers until result (CL and CM) convergence is achieved. For simulation with number of panels variation, wing whose airfoil NACA 4412 at zero angle of attack is used. The detail geometry parameters are wing root chord is 1 m, wing tip chord is 0.6 m, wing span is 10 m, and aft sweep back chord-wise direction is 0.1 m. The simulation is only half wing span. Five simulation with different number of panels following Table 4.1 are conducted.

Case	Chord-wise Panels	Span-wise Panels
1	20	3
2	30	5
3	40	7
4	50	9
5	60	11

Table 4.1: Number of Panels Variation

The pressure coefficient at each panel for case 1 and case 4 are plotted in Figure. 4.1 and Figure. 4.2 respectively. From these figures, the wing geometry could be seen clearly and also the variation of air flow along chord-wise and span-wise direction could be observed. For simulation with smaller number of panels used, it is obvious that the result will be lack of detail and there is some numerical error occurs. We need to determine when the convergence value achieved, and the increment of number of panels is not very significant to the result. For wing with airfoil NACA4412, there is lift generated at zero angle of attack configuration. The pressure coefficient at the upper surface is lower than at the lower surface.

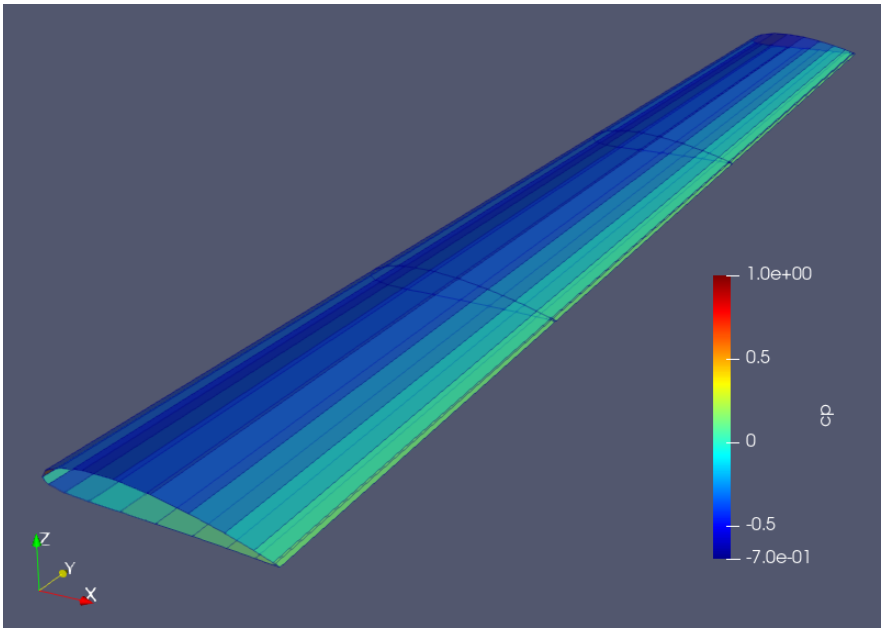


Figure 4.1: Pressure Coefficient Distribution (20 Chord-wise, 6 Span-wise Panels)

The result in Figure 4.2 is much smoother that the result in Figure. 4.1 and must be more accurate. Overall, the general pressure coefficient distribution from both figures are similar and tend to be a correct simulation where pressure coefficient is very high at the leading edge due to the stagnation point, and the upper surface has lower pressure coefficient compared to the lower surface, which is the case when lift is generated.

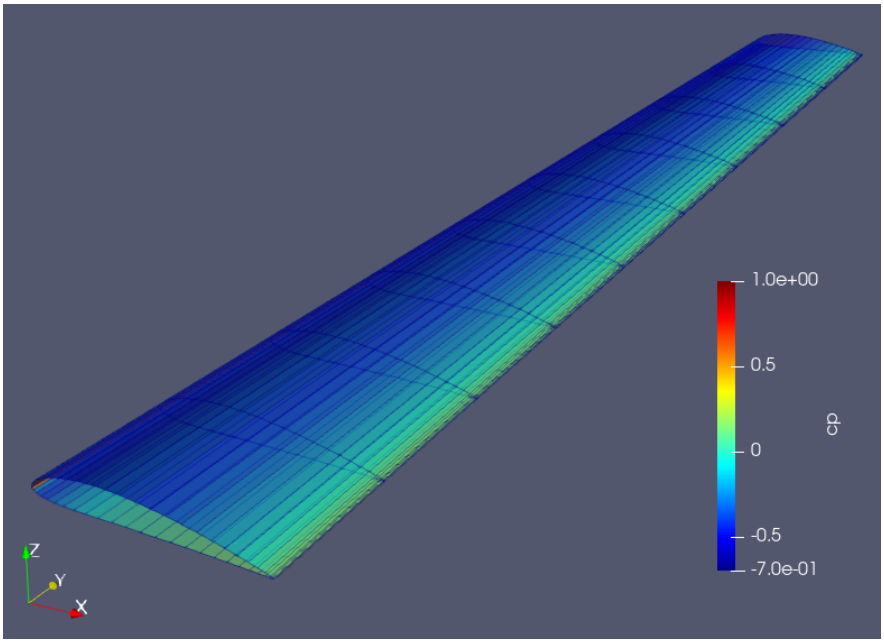


Figure 4.2: Pressure Coefficient Distribution (50 Chord-wise 18 Span-wise Panels)

After five simulations were conducted, then for each case the value of lift coefficient (CL), moment coefficient (CM), and computation time are plotted in Figure. 4.3, Figure. 4.4, and Figure. 4.5 respectively.

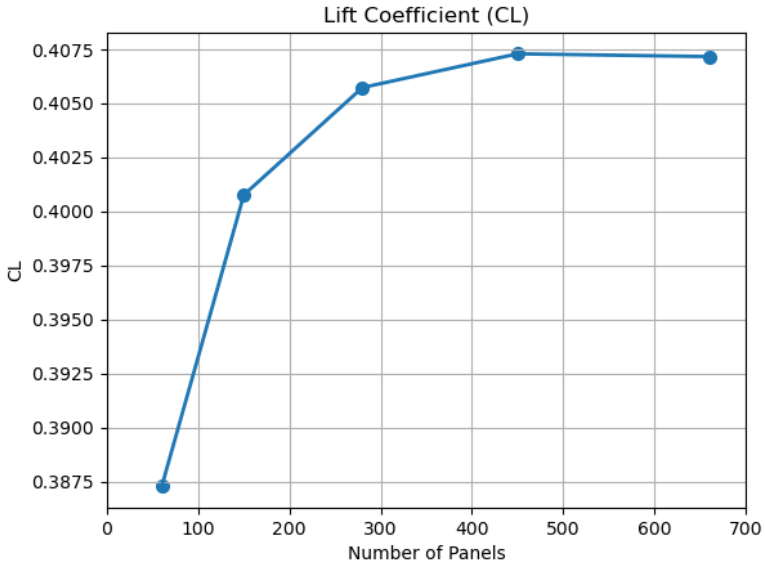


Figure 4.3: Lift Coefficient with Number of Panels Variation

The convergence trend could be seen from both lift coefficient and moment coefficient graph, the values keep increasing until relatively constant at case 4 and case 5.

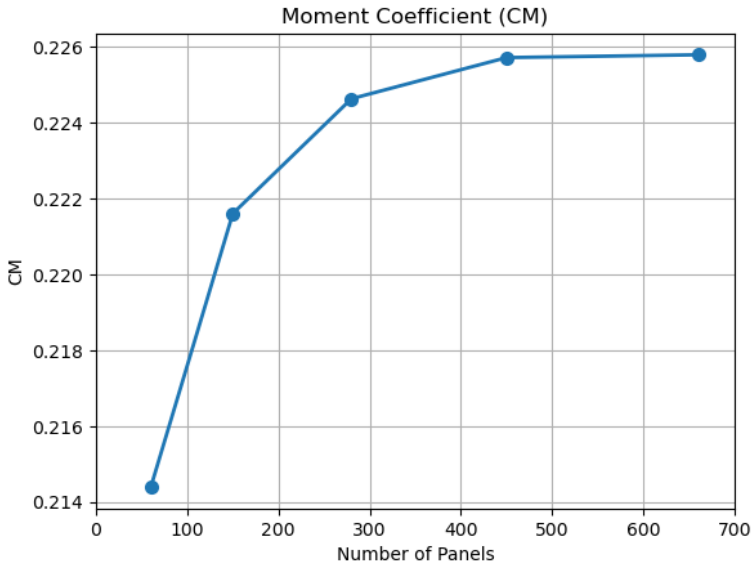


Figure 4.4: Moment Coefficient with Number of Panels Variation

The computation time is growing at exponential trend when the number of panels is increased. Because convergence has been achieved at case 4 and case 5, then conducting the simulation using number of panels base on case 4 which is 50 panels chord-wise and 18 panels span-wise will be enough to get accurate result. Increasing the number of panels beyond this case will increase the computation time enormously but the simulation result will be the same.

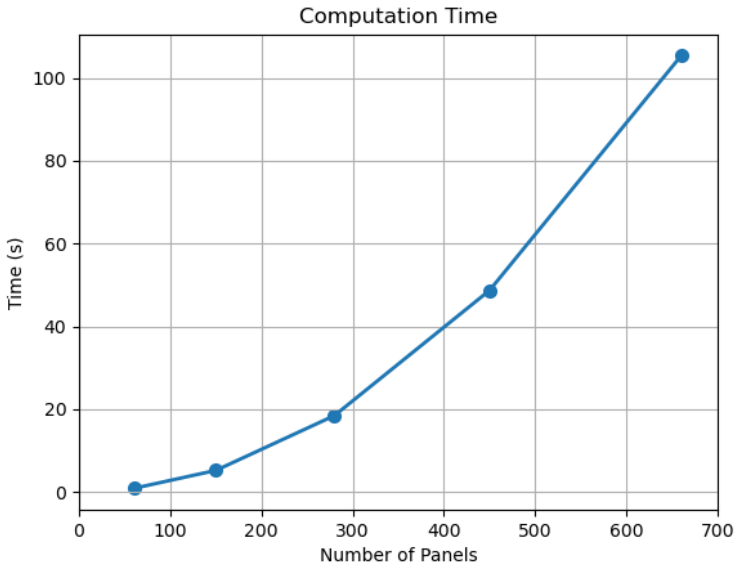


Figure 4.5: Computational Time with Number of Panels Variation

4.2 Wake Angle Variation

One of important parameters for 3D Panel Method is wake geometry. Wake geometry define influence of doublets on wake to satisfy Kutta Condition on trailing edge. For simplicity, the wake is modelled as very long panel with angle that can be varied. There are two variation of wake angle: type 1 with zero angle and type 2 with median angle of trailing edge like Figure 4.6. Geometry parameters for this study is same with geometry on study of number of panels variation with case 4.

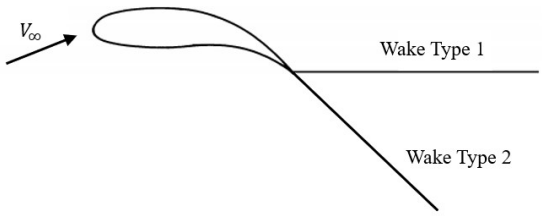


Figure 4.6: Wake Geometry Illustration

Based on Table 4.2, we can see that different wake angle can give different results on aerodynamic coefficients even though the difference is small. Figure 4.7 and 4.8 show the difference between two type of wake is small. If we compare it with well-known software of 3D Panel Method, XFLR5, we can see that relative errors is around 1 until 2% and wake type 2 gives relative error higher than wake type 1. This numerical errors can be minimized if we use higher order of panel method and using dynamic wake iteration until converge to one type of wake geometry.

Angle of Attack	Coefficient	XFLR5	Wake Type 1	Wake Type 2	Error Wake Type 1	Error Wake Type 2
0°	CL	0.3978	0.4073	0.4085	2.39%	2.69%
	CM	-0.2218	-0.2257	-0.2263	1.77%	2.05%
1°	CL	0.4930	0.5034	0.5050	2.12%	2.44%
	CM	-0.2507	-0.2549	-0.2557	1.65%	1.97%
2°	CL	0.5879	0.5993	0.6013	1.93%	2.27%
	CM	-0.2796	-0.2839	-0.2849	1.55%	1.90%
3°	CL	0.6826	0.6947	0.6971	1.77%	2.13%
	CM	-0.3083	-0.3128	-0.3140	1.46%	1.84%

Table 4.2: Results of Wake Angle Variation

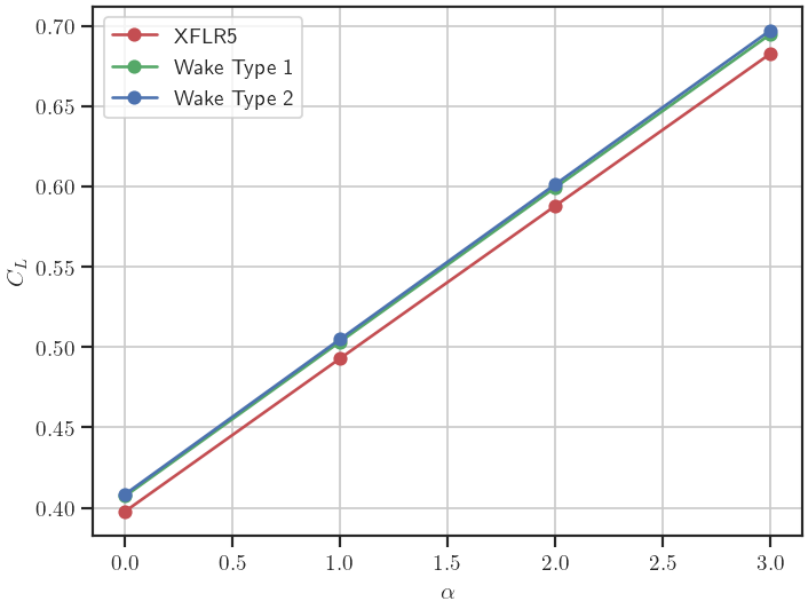


Figure 4.7: Curve of C_L vs α for Wake Geometry Variaton

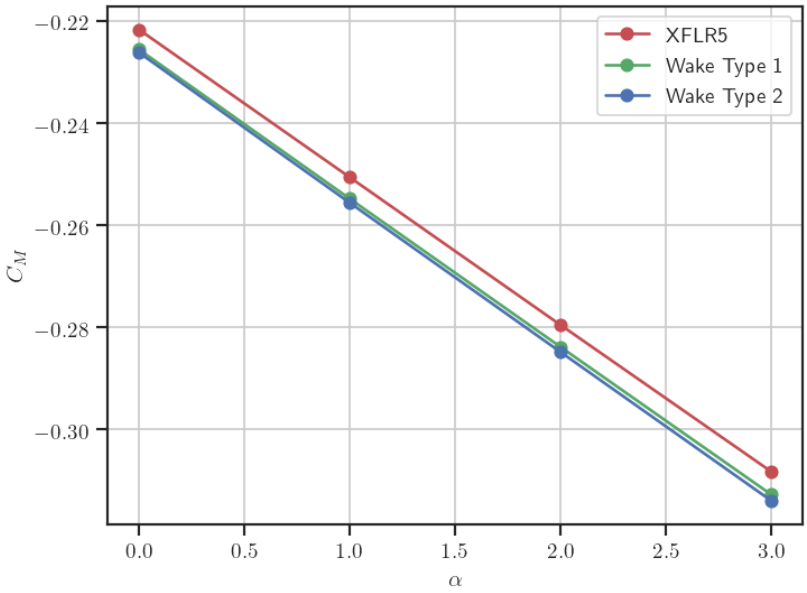


Figure 4.8: Curve of C_M vs α for Wake Geometry Variaton

Chapter 5 Conclusion

5.1 Conclusion

There are some conclusions from this project work.

1. Inviscid flow around wing can be analyzed easily using Panel Method and get distribution of pressure on wing which is the important part of wing analysis. This method also computationally efficient than FVM or FDM that need all the fluid to be discretized.

2. Increasing number of panel make the computation time is growing at exponential trend and can be seen at Figure 4.5. But at one number panel, the results already convergent and can be seen at Figure 4.3 and 4.4. For study on this project, 450 panels on half wing already gives convergent result on aerodynamic coefficients.

3. Different wake angle will give different results even though the difference is small. If we compare it with XFLR5, the difference of every wake angle with the results from XFLR5 only different by around 1% until 2% and can be seen at Figure 4.7 and 4.8.

5.2 Future Works

Future works for this project are suggested as follows:

- Derive influence matrix of panel using higher order strength like 1st order or 2nd order function of polynomials.

- Derive influence matrix of panel with different panel geometry like triangle.

- Make algorithm for dynamic wake iteration to find convergent wake geometry.

References

[1] K. Joseph and A. Plotkin. *Low Speed Aerodynamics*, Cambridge University Press, 2001.

[2] J. D. Anderson. *Fundamentals of Aerodynamics 5th Edition*. McGraw-Hill, New York, 2010.

[3] A. M. Kuethe and C. Y. Chow. *Foundation of Aerodynamics Bases of Aerodynamic Design*, John Wiley and Sons, Canada, 1998.

[4] F. T. Johnson. *A General Panel Method for the Analysis and Design of Arbitrary Configurations in Incompressible Flows*. NASA Contractor Report 3079, Seattle, 1980.

Source Code

The simulation code was written using Python. The main program is inside the file *Main.py*. Many helper functions were created to perform the calculation. In general could be divided into:

1. Grid generation inside *Grid.py*
    - to load airfoil (*Load.py*)
    - to create panel grid (*Panel.py*)
    - to convert to local coordinate (*Convert.py*)
2. Aerodynamic calculation inside *Compute_A_RHS.py*
    - to calculate influence coefficient ($Influence.py$)
3. Aerodynamic force calculation inside *Force_Calculation.py*

Additional code for visualization proposes was written in C++ to make .vtu file format which can be read by ParaView (*print_vtu.cpp*).

```
# Main.py

import numpy as np
from Grid import Grid
from Compute_A_RHS import Compute_A_RHS
from Force_Calculation import Force_Calculation

import time

# starting time
start = time.time()
# %% INPUT
```

```
input = [0.0,          # alpha
         1.0,          # cRoot (root chord)
         0.6,          # cTip  (tip chord)
         0.1,          # xTip  (aft sweep of tip)
         0.0,          # zTip  (aft sweep of tip)
         10.0,         # b     (span)
         1.0,          # vInf  (freestream vel.)
         int(41),    # iB1   (number of airfoil points)
         int(7)]       # jB    (number of panel spanwise)

input.append(input[5]*100)                          # dxW  (length
                                    of wake)
input.append(input[6]*np.cos(input[0]*np.pi/180)) # uInf (v in x
                                    dir.)
input.append(input[6]*np.sin(input[0]*np.pi/180)) # wInf (v in z
                                    dir.)

# Airfoil name
input.append('naca4412')
'''
flagStl = [0,            # Turn on/off Streamline Calculation (on=
                                    1,off=0)
           21,           # nGridX (point Grid X)
           21,           # nGridZ (point Grid Z)
           [-0.5, 1.5], # xVals (Range Grid x)
           [-0.5, 0.5], # zVals (Range Grid z)
           10]           # Percentage of position y to half span
'''
# Extract iB1 and jB
iB1    = input[7]
jB     = input[8]

# %% GRID GENERATION

# Grid generation using function
qF, qC, r, S, sig, cR = Grid(input)

iB  = iB1 - 1 # number of wing panel chordwise
iB2 = iB1 + 1 # number of points chordwise
jB1 = jB  + 1 # number of points spanwise

# %% AERODYNAMIC CALCULATIONS
```

```
# Compute matrix A and vector RHS using function
A, rhs = Compute_A_RHS(input,qF,qC,r,S,sig,cR)

# Solve the system of equation for doublet strength mu
mu1 = np.linalg.solve(A,rhs)

# Reshape vector mu1 into matrix of panel
mu  = np.zeros([iB1,jB])
mu[0:iB,:] = np.reshape(mu1,(iB,jB))

# Compute doublet strength for wake
for j in range(jB):
    mu[iB,j] = mu[0,j] - mu[iB-1,j]

# %% FORCE CALCULATIONS

# Calculate force coefficient for wing using function
Cp, Cl, Cd, Cm = Force_Calculation(input,qF,qC,r,S,mu)

print('CL :',Cl)
print('CM :',Cm)

# end time
end = time.time()

# total time taken
print(f"Runtime of the program is {end - start}")

#%% Visualization

file1 = open("Panel.dat","w")
#file1.write("Hello \n")
file1.write("%d %d\n" % (iB1*jB1,iB*jB)) #number of nodes &
                                    number of elements

#print nodes
index = 0;
nodes = np.zeros((iB1*jB1,3))
for i in range(iB1):
    for j in range(jB1):
        file1.write("%d " %(index+1))
        for k in range(3):
            nodes[index][k] = qF[i][j][k]
```

```
            file1.write("%f " % (nodes[index][k]))
        file1.write("\n")
        index += 1

#print elements(panel)
index = 0;
elem = np.zeros((iB*jB,4))
for i in range(iB):
    for j in range(jB):
        elem[index][0] = i*jB1 + j
        elem[index][1] = i*jB1 + (j+1)
        elem[index][2] = (i+1)*jB1 + (j+1)
        elem[index][3] = (i+1)*jB1 + j

        file1.write("%d %d %d %d %d %d\n" % (index+1,9,elem[index
                                            ][0]+1,elem[index][1]+1,
                                            elem[index][2]+1,elem[
                                            index][3]+1))
        index += 1

#print data at each element
index = 0;
data = np.zeros((iB*jB,2)) #cp and area
for i in range(iB):
    for j in range(jB):
        data[index][0] = Cp[i][j]
        data[index][1] = S[i][j]

        file1.write("%d %f %f\n" % (index+1,data[index][0],data[
                                            index][1]))
        index += 1
file1.close()
```

```
#Grid.py

import numpy as np
import math as m
from scipy.interpolate import CubicSpline
from Load import Load_Airfoil
from Panel import Panel
from Convert import Convert

def Grid(input):
```

```
# To generate grid of wing based on input

# Extract input vars.
cRoot = input[1]
cTip  = input[2]
xTip  = input[3]
zTip  = input[4]
b     = input[5]
iB1   = input[7]
jB    = input[8]
dxW   = input[9]
uInf  = input[10]
wInf  = input[11]

# Load airfoil coordinates
rA = Load_Airfoil(input[12])

# Find LE
dist = 0
for i in range(len(rA[:,0])):
    temp = (rA[i,0]-rA[0,0])**2 - (rA[i,1]-rA[0,1])**2
    dist = max(dist,temp)
    if (dist != temp):
        nLE = i - 1
        break

# Transform x coordinate to psi
ch = np.abs(rA[nLE,0] - rA[0,0])
psi = np.zeros(len(rA[:,0]))
for i in range(len(rA[:,0])):
    phiA   = m.acos((rA[i,0]-0.5*ch)/(0.5*ch))
    if (i <= nLE):
        psi[i] = phiA
    elif (i >= nLE):
        psi[i] = 2*np.pi - phiA

# Cubic spline yA as function of psi
cs = CubicSpline(psi,rA[:,1])

# Define new psi
psiA = np.linspace(0,2*np.pi,iB1)

# Compute new airfoil coordinate
```

```
rB      = np.zeros([iB1,2])
rB[:,0] = 0.5*ch*(1+np.cos(psiA))
rB[:,1] = cs(psiA)

# Check for direction of points
area = 0
for i in range(iB1-1):
    area = area + 0.5*(rB[i+1,0]-rB[i,0])*(rB[i+1,1]+rB[i,1])

# If panels are CCW, flip them
if (area < 0):
    rB = np.flipud(rB)

# Define number of panels
iB  = iB1 - 1
iB2 = iB1 + 1
jB1 = jB  + 1

# Initialization of qF, qC, r (c, t, v), S, sig
qF  = np.zeros([iB2,jB1,3])
qC  = np.zeros([iB1,jB,3])
r   = np.zeros([iB1,jB,9])
S   = np.zeros([iB1,jB])
sig = np.zeros([iB1,jB])
cR  = np.zeros([iB1,jB,12])

# Move airfoil data from rB to qF
qF[0:iB1,0,0] = rB[:,0]
qF[0:iB1,0,2] = rB[:,1]

# Close TE point
qF[0,0,2] = 0.5*(qF[0,0,2]+qF[iB,0,2])
qF[iB,0,2] = qF[0,0,2]

# Calculate panel cornerpoints
for j in range(jB1):

    # Define y, LE position, and chord
    y    = (0.5*b/jB)*j
    dxLE = xTip*2*y/b
    dzLE = zTip*2*y/b
    chrd = cRoot - (cRoot - cTip)*2*y/b
```

```
        # Define cornerpoints of panels on wing
        for i in range(iB1):
            qF[i,j,0] = qF[i,0,0]*chrd + dxLE
            qF[i,j,1] = y
            qF[i,j,2] = qF[i,0,2]*chrd + dzLE

        # Define wake cornerpoints
        qF[iB1,j,0]   = qF[iB,j,0] + dxW
        qF[iB1,j,1:2] = qF[iB,j,1:2]

    # Define wing collocation points
    for j in range(jB):
        for i in range(iB1):
            qC[i,j,:] = (qF[i,j,:]+qF[i,j+1,:]+qF[i+1,j+1,:]+qF[i
                                                +1,j,:])/4.

            # Call panel subroutines
            r[i,j,:], S[i,j] = Panel(qF[i,j,:],qF[i+1,j,:],qF[i,j
                                                +1,:],qF[i+1,j+1,:])

            # Calculate sigma distribution
            sig[i,j] = r[i,j,6]*uInf + r[i,j,8]*wInf

            # Transform cornerpoints into panel frame of ref.
            cR[i,j,0:3]  = Convert(qC[i,j,:],qF[i,j,:],r[i,j,:])
            cR[i,j,3:6]  = Convert(qC[i,j,:],qF[i+1,j,:],r[i,j,:]
                                                )
            cR[i,j,6:9]  = Convert(qC[i,j,:],qF[i+1,j+1,:],r[i,j,
                                                :])
            cR[i,j,9:12] = Convert(qC[i,j,:],qF[i,j+1,:],r[i,j,:]
                                                )

    return qF, qC, r, S, sig, cR
```

```
#Load.py

import numpy as np
import os

def Load_Airfoil(fileName):
    # To load airfoil coordinate from .dat file in repo
                                        Airfoil_DAT_Selig
```

```
    # Define the number of header lines in the file
    hdrlns = 1
    if (fileName == 'nasasc2-0714'):
        hdrlns = 3
    elif (fileName == 's1020'):
        hdrlns = 2

    # Load the data from the text file
    crpth = os.path.dirname(__file__)
    flpth = crpth +"/Airfoil_DAT_Selig/"
    flnm = flpth + fileName + ".dat"
    dataBuffer = np.loadtxt(flnm,skiprows=hdrlns)

    return dataBuffer
```

```
#Panel.py

import numpy as np
from numpy import linalg as LA

def Panel(r1,r2,r3,r4):
    # To compute basis of the panel based on coordinate of
                                        cornerpoints

    # Calculate vector A and chordwise vector (c)
    A = np.zeros(3)
    A = ((r2-r1) + (r4-r3))/2. # A is average chordwise vector of
                                        two sides
    c = A/LA.norm(A)           # c is normalization of A

    # Calculate another vector on the plane (B)
    B = r4 - r1

    # Calculate normal vector of the plane (v)
    v = np.cross(c,B)  # v is cross product of c and B
    v = v/LA.norm(v)   # normalization of v

    # Calculate tangential vector on the plane (t)
    t = np.cross(v,c)  # t is cross product of v and c

    # Calculate panel area (S)
    E  = r3 - r1        # define first vector in edge
    F  = r2 - r1        # define second vector in edge
```

```
    E1 = np.cross(B,E)  # cross product of first half-triangle
    F1 = np.cross(F,B)  # cross product of second half-triangle
    S1 = LA.norm(E1)    # area of first parallelogram
    S2 = LA.norm(F1)    # area of second parallelogram
    S  = 0.5*(S1 + S2)  # area of panel is half of area of
                                    parallelogram

    # Save c, t, v in r
    r = np.zeros(9)
    r[0:3] = c
    r[3:6] = t
    r[6:9] = v

    return r, S
```

```
#Convert.py

import numpy as np

def Convert(r0,rB,r):
    # To convert rB into panel frame of ref. (r0 and r)
    #
    # r0 is origin coordinate (frame of ref.)
    # rB is coordinate to be converted
    # r  is basis of panel in array 1D

    r  = np.reshape(r,(3,3))  # reshape r to 3x3
    rP = np.matmul(r,(rB-r0)) # basis * relative coordinate

    return rP
```

```
#Compute_A_RHS.py

import numpy as np
from Convert import Convert
from Influence import Influence

def Compute_A_RHS(input,qF,qC,r,S,sig,cR):
    # To compute matrix A (influence matrix) and vector RHS
    # Input is wing geometry

    # Extract input vars.
    cRoot = input[1]
```

```
cTip   = input[2]
xTip   = input[3]
zTip   = input[4]
b      = input[5]
vInf   = input[6]
iB1    = input[7]
jB     = input[8]
dxW    = input[9]
uInf   = input[10]
wInf   = input[11]

# Number of wing panel chordwise
iB     = iB1 - 1

# Initialization of matrix and vector
A     = np.zeros([iB*jB,iB*jB])
rhs   = np.zeros(iB*jB)
dDub  = np.zeros(jB)
qCi   = np.zeros(3)

k     = - 1
for i in range(iB):
    for j in range(jB):
        k  = k + 1 # Control panel counter
        l  = - 1
        rh = 0
        # Define image of control panel
        qCi[:] = qC[i,j,:]
        qCi[1] = - qCi[1]
        for i1 in range(iB):
            for j1 in range(jB):
                l = l + 1 # Influence panel counter
                if (i1 == 0):
                    # Convert control panel to wake panel
                                                    frame of
                                                     ref.
                    rC = Convert(qC[iB,j1,:],qC[i,j,:],r[iB,
                                                    j1,:])

                    # Calculate wake influence
                    wDub, dSig = Influence(rC,cR[iB,j1,:])

                    # Convert image to wake panel frame of
```

```
                                                    ref.
                    rC = Convert(qC[iB,j1,:],qCi,r[iB,j1,:])

                    # Calculate image influence
                    wDub1, dSig = Influence(rC,cR[iB,j1,:])

                    # Save wake doublet influence
                    dDub[j1] = wDub + wDub1
                    dMu2 = dDub[j1]
                else:
                    dMu2 = 0.

                if (i1 == (iB-1)):
                    dMu2 = - dDub[j1]

                # Convert control panel to influence panel
                                                frame of ref
                                                .
                rC = Convert(qC[i1,j1,:],qC[i,j,:],r[i1,j1,:]
                                                )

                # Calculate panel influence
                dMu, dSig = Influence(rC,cR[i1,j1,:])
                if (i1 == i and j1 == j):
                    dMu = - 0.5

                # Convert image to influence panel frame of
                                                ref.
                rC = Convert(qC[i1,j1,:],qCi,r[i1,j1,:])

                # Calculate panel influence
                dMu1, dSig1 = Influence(rC,cR[i1,j1,:])

                # Influence matrix coefficient
                A[k,l] = dMu + dMu1 - dMu2
                rh     = rh + (dSig + dSig1)*sig[i1,j1]

        rhs[k] = rh

    return A, rhs
```

```
#Influence.py
import numpy as np
```

```
import math as m

def Influence(rC,cR):
    # To compute influence coefficient of panel (cR) to point rC

    eps = 0.000001
    rF  = np.reshape(cR,(4,3))

    # Calculate distance for 'Influence Calculation'
    r  = np.zeros(4)
    e  = np.zeros(4)
    h  = np.zeros(4)
    d  = np.zeros(4)
    for i in range(4):
        r[i] = np.sqrt((rC[0]-rF[i,0])**2 + (rC[1]-rF[i,1])**2 +
                                            rC[2]**2)
        e[i] = (rC[0]-rF[i,0])**2 + rC[2]**2
        h[i] = (rC[0]-rF[i,0])*(rC[1]-rF[i,1])
        if (i != 3):
            d[i] = np.sqrt((rF[i+1,0]-rF[i,0])**2 + (rF[i+1,1]-rF
                                                [i,1])**2)
        else:
            d[i] = np.sqrt((rF[0,0]-rF[i,0])**2 + (rF[0,1]-rF[i,1
                                                ])**2)

    # Calculate 'Influence Component'
    s = np.zeros(4)
    q = np.zeros(4)
    A = 0
    B = 0
    for i in range(4):
        if (d[i] < eps):
            s[i] = 0
            q[i] = 0
        else:
            if (i != 3):
                F    = (rF[i+1,1]-rF[i,1])*e[i] - (rF[i+1,0]-rF[i
                                                ,0])*h[i]
                G    = (rF[i+1,1]-rF[i,1])*e[i+1] - (rF[i+1,0]-rF
                                                [i,0])*h[i+1]
                num  = rC[2]*(rF[i+1,0]-rF[i,0])*(F*r[i+1]-G*r[i]
                                                )
                den  = (rC[2]*(rF[i+1,0]-rF[i,0]))**2*r[i]*r[i+1]
```

```
                                                + F*G
                q[i] = m.atan2(num,den)
                num  = (rC[0]-rF[i,0])*(rF[i+1,1]-rF[i,1])
                num  = num - (rC[1]-rF[i,1])*(rF[i+1,0]-rF[i,0])
                s[i] = (num/d[i])*m.log((r[i]+r[i+1]+d[i])/(r[i]+
                                                r[i+1]-d[i]))
            else:
                F = (rF[0,1]-rF[i,1])*e[i] - (rF[0,0]-rF[i,0])*h[
                                                i]
                G = (rF[0,1]-rF[i,1])*e[0] - (rF[0,0]-rF[i,0])*h[
                                                0]
                num  = rC[2]*(rF[0,0]-rF[i,0])*(F*r[0]-G*r[i])
                den  = (rC[2]*(rF[0,0]-rF[i,0]))**2*r[i]*r[0] + F
                                                *G
                q[i] = m.atan2(num,den)
                num  = (rC[0]-rF[i,0])*(rF[0,1]-rF[i,1])
                num  = num - (rC[1]-rF[i,1])*(rF[0,0]-rF[i,0])
                s[i] = (num/d[i])*m.log((r[i]+r[0]+d[i])/(r[i]+r[
                                                0]-d[i]))

        A = A + q[i]
        B = B + s[i]

    A = - A/(4*np.pi)
    if (abs(rC[2]) < eps):
        A = 0

    B = - B/(4*np.pi) - rC[2]*A

    return A, B
```

```
#Force_Calculation.py

import numpy as np

def Force_Calculation(input,qF,qC,r,S,mu):
    # To calculate force on wing

    # Extract input vars.
    cRoot = input[1]
    cTip  = input[2]
    xTip  = input[3]
    zTip  = input[4]
```

```
b      = input[5]
vInf   = input[6]
iB1    = input[7]
jB     = input[8]
dxW    = input[9]
uInf   = input[10]
wInf   = input[11]

# Number of wing panel chordwise
iB  = iB1 - 1

# Initialization of array
Cp = np.zeros([iB,jB])
dL = np.zeros([iB,jB])
dD = np.zeros([iB,jB])
CL = 0
CD = 0
CM = 0

rF = np.zeros(3)
rR = np.zeros(3)
d2 = np.zeros(3)
d3 = np.zeros(3)
dr = np.zeros(3)

# Compute velocity using finite difference
for j in range(jB):
    for i in range(iB):
        i1 = i - 1
        i2 = i + 1
        j1 = j - 1
        j2 = j + 1
        if (i == 0):
            i1 = 0
        if (i == (iB-1)):
            i2 = iB - 1
        if (j == 0):
            j1 = 0
        if (j == (jB-1)):
            j2 = jB - 1

        rF[:] = 0.5*(qF[i+1,j,:] + qF[i+1,j+1,:])
        rR[:] = 0.5*(qF[i,j,:] + qF[i,j+1,:])
```

```
            d2[:] = qC[i2,j,:] - rF
            d3[:] = qC[i1,j,:] - rR
            dl1   = np.linalg.norm(rF-rR)
            dl2   = np.linalg.norm(d2)
            dl3   = np.linalg.norm(d3)
            dll   = dl1 + dl2 + dl3
            if (i == 0):
                dll = dl1/2 + dl2
            if (i == (iB-1)):
                dll = dl1/2 + dl3
            ql    = - (mu[i2,j] - mu[i1,j])/dll
            dr[:] = qC[i,j2,:] - qC[i,j1,:]
            delR  = np.linalg.norm(dr)
            qm    = - (mu[i,j2] - mu[i,j1])/delR
            ql    = ql + qm*(dr[0]**2 + dr[2]**2)/delR
            qm    = qm*(dr[1]**2 + dr[2]**2)/delR
            qInf  = uInf*r[i,j,8] - wInf*r[i,j,6]
            Cp[i,j] = 1.0 - ((qInf + ql)**2 + qm**2)/(vInf**2)
            dL[i,j] = - Cp[i,j]*S[i,j]*r[i,j,8]
            dD[i,j] = Cp[i,j]*S[i,j]*r[i,j,6]
            CL = CL + dL[i,j]
            CD = CD + dD[i,j]
            CM = CM + dL[i,j]*qC[i,j,0]

    # Compute mean aerodynamic chord
    lam   = cTip/cRoot
    cMean = (2*cRoot/3)*(1+lam+lam**2)/(1+lam)

    # Compute force coefficient
    CL = CL/(0.25*(cRoot+cTip)*b)
    CD = CD/(0.25*(cRoot+cTip)*b)
    CM = CM/(0.25*(cRoot+cTip)*b*cMean)

    return Cp, CL, CD, CM
```

```
//print_vtu.cpp

#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <stdlib.h>
#include <math.h>
```

```
using namespace std;

/*definition of a node*/
struct Node
{
  double pos[3];
  Node(double x, double y, double z) {pos[0]=x;pos[1]=y;pos[2]=z;
                                      }
};

/*definition of a quad*/
struct Quad
{
  int con[4];
  double area;
  double Cp;
  Quad(int n1, int n2, int n3, int n4) {con[0]=n1;con[1]=n2;con[2
                                      ]=n3;con[3]=n4;}
};

/*definition of a surface*/
struct Panel
{
  vector <Node> nodes;
  vector <Quad> elements;
};

void Output(Panel &panel);

int main()
{
    /*open file*/
  ifstream in("Panel.dat");
  if (!in.is_open()) {cerr<<"Failed to open input file";return -1
                                      ;}

  /*read number of nodes and elements*/
  int n_nodes, n_elements;
  in>>n_nodes>>n_elements;
  cout<<"Mesh contains "<<n_nodes<<" nodes and "<<n_elements<<"
                                      elements"<<endl;
```

```
/*instantiate panel*/
Panel panel;

/*read the nodes*/
for (int n=0;n<n_nodes;n++)
{
  int index;
  double x, y, z;

  in >> index >> x >> y >> z;
  if (index!=n+1) cout<<"Inconsistent node numbering"<<endl;

  panel.nodes.emplace_back(x,y,z);
}

//read elements, this will also contain edges
for (int e=0;e<n_elements;e++)
{
  int index, type;
  int n1, n2, n3, n4;

  in >> index >> type;

  if (type!=9) {string s; getline(in,s);continue;}

  in >> n1 >> n2 >> n3 >> n4;

  panel.elements.emplace_back(n1-1, n2-1, n3-1, n4-1);  //if
                                nodes indexing starts from 1
}

//read element data
for (int e=0;e<n_elements;e++)
{
  int index;
  double Cp,Area;

  in >> index;

  in >> Cp >> Area;

  panel.elements[e].Cp = Cp;
  panel.elements[e].area = Area;
```

```
  }

    Output(panel);
}


void Output(Panel &panel)
{
    ofstream out("Panel.vtu");
  if (!out.is_open()) {cerr<<"Failed to open output file "<<endl;
                                exit(-1);}

  /*header*/
  out<<"<?xml version=\"1.0\"?>\n";
  out<<"<VTKFile type=\"UnstructuredGrid\" version=\"0.1\"
                                byte_order=\"LittleEndian\">\n";
  out<<"<UnstructuredGrid>\n";
  out<<"<Piece NumberOfPoints=\""<<panel.nodes.size()<<"\"
                                NumberOfVerts=\"0\"
                                NumberOfLines=\"0\" ";
  out<<"NumberOfStrips=\"0\" NumberOfCells=\""<<panel.elements.
                                size()<<"\">\n";

  /*points*/
  out<<"<Points>\n";
  out<<"<DataArray type=\"Float32\" NumberOfComponents=\"3\"
                                format=\"ascii\">\n";
  for (Node &node: panel.nodes)
    out<<node.pos[0]<<" "<<node.pos[1]<<" "<<node.pos[2]<<"\n";

  out<<"</DataArray>\n";
  out<<"</Points>\n";

  /*Cells*/
  out<<"<Cells>\n";
  out<<"<DataArray type=\"Int32\" Name=\"connectivity\" format=\"
                                ascii\">\n";
  for (Quad &quad: panel.elements)
    out<<quad.con[0]<<" "<<quad.con[1]<<" "<<quad.con[2]<<" "<<
                                quad.con[3]<<"\n";
  out<<"</DataArray>\n";

  out<<"<DataArray type=\"Int32\" Name=\"offsets\" format=\"ascii
```

```
                                    \">\n";
  for (int e=0; e<panel.elements.size();e++)
    out<<(e+1)*4<<" ";
  out<<"\n";
  out<<"</DataArray>\n";

  out<<"<DataArray type=\"UInt8\" Name=\"types\" format=\"ascii
                                    \">\n";
  for (int e=0; e<panel.elements.size();e++)
    out<<"9 ";
  out<<"\n";
  out<<"</DataArray>\n";
  out<<"</Cells>\n";

  /*save Cp data*/
  out<<"<CellData Scalars=\"properties\">\n";
  out<<"<DataArray type=\"Float32\" Name=\"cp\" format=\"ascii
                                    \">\n";
  for (Quad &quad:panel.elements)
    out<<quad.Cp<<" ";
    //out<<1<<" ";
  out<<"\n";
  out<<"</DataArray>\n";

  /*save area data*/
  out<<"<DataArray type=\"Float32\" Name=\"panel_area\" format=\"
                                    ascii\">\n";
  for (Quad &quad:panel.elements)
    out<<quad.area<<" ";
    //out<<1<<" ";
  out<<"\n";
  out<<"</DataArray>\n";

  out<<"</CellData>\n";

  out<<"</Piece>\n";
  out<<"</UnstructuredGrid>\n";
  out<<"</VTKFile>\n";

  out.close();
}
```