Computational Fluid Dynamics of 2D Phenomena

CS 50100 Computing for Science and Engineering
Final Report

Stephen Hannah (Leader)
Luis G. Gutierrez
Matthew Liu
Natasha Singh
Christopher Zoller

Purdue University
December 4th, 2020

Contents

1 Introduction

Understanding fluid phenomena is crucial in a wide range of fields and applications, from energy generation systems —like turbines and engines—, weather and atmospheric processes, design of transportation vehicles, to flow of biological fluids in living organisms. Although fluid flow applications are ubiquitous, analytical solutions to fluid problems are limited; only a few canonical examples exist where the governing equations of fluid flow can be solved directly [1]. As a result of this limitation, computational and numerical approaches gained importance in the realm of fluids engineering. Computational Fluid Dynamics (CFD) is now one the pillars in the analysis and design of fluid systems as demonstrated in the works of [2], [3], and [4].

One of the canonical problems in fluid dynamics is the analysis of inviscid flow around a 2D object using potential flow. Undergraduate mechanical and aerospace engineering curricula commonly cover the topic of potential flow from the perspective of analytical solutions. However, the analytical tools used to solve potential flow tend to have a steep learning curve which can slow down student learning and hinder the engagement. The use of CFD and computation tools can assist introducing students to the concepts of fluid mechanics and potential flow through a different perspective. CFD has been proposed as pedagogical tools to increase student interaction with the material as well as intellectual curiosity and discovery [2].

1.1 Related Work

The inspiration to creating the code comes from XFLR, a well-known and widely-used aerodynamics tool for model airplane design. XFLR features a variety of capabilities in designing basic model planes from airfoils and bodies, and uses panel methods to quickly calculate the flight profile of a given model. It can also see viscous effects and perform stability analysis, all of which are advanced features not included in this code. To add something unique, the ability to draw a 2D geometry freehand was implemented in the code. XFLR allows custom airfoil creation with the use of splines, but does not have a freehand capability for more obscure geometries. Creating code similar to this that solves the incompressible, irrotational, and inviscid flow is commonly featured in graduate-level fluid mechanics courses, but the extra step was taken to create a GUI for potential users.

1.2 Contribution

The two main sources of contribution for this work came from Aeropython [5] and Josh the Engineer [6]. These two sources provided the necessary detailed derivations and example Python code that could be understood and used for creating a code that used panel methods to produce aerodynamic outcomes for the user. Initially Aeropython was used in order to understand what the source panel method is and how it could be coded into python. Upon further research a more comprehensive explanation of the derivations necessary to evaluate difficult equations relating to panel methods was found with Josh the Engineer [6] Here a step by step derivation of every required equation for both source and vortex panel methods were played out in a series of videos on YouTube. Using this, along with code written to pair with the videos, a python program was able to be written that allowed for both the vortex and source panel method to be used simultaneously on any geometry placed into the GUI.

1.3 Problem Statement

In the study of aerodynamics and fluid dynamics, new educational resources to aid in teaching these topics are constantly improving. Equations to solve potential flow expressions have been around for decades, but recently the increase in computing power in new technology have enabled developers to create software to visualize flow around objects. Lots of the software on the web for CFD has predefined inputs and can take time to run the simulations. For new students to these areas, there is room for developing tools that provide a quick way to visualize and measure the performance of creative shapes.

The objective is to develop an interactive tool for students in the aerodynamics and fluid dynamics fields to take advantage of in their studies. By using this CFD tool, the students can input predefined shapes as well as hand drawn prototypes into the interface to enable them to easily test their new ideas. Once the shape is generated, the user can visualize their simulation with contour and streamline plots. Important aerodynamic parameters are then output back to the user based on their simulation. This structure gives the user an efficient method of simulating multiple shapes and comparing their performances.

1.4 Organization

The succeeding sections of the report go into elaborating on the methods and key approaches for solving the problem statement of this report along with providing the main results of the simulations. The second section specifically addresses the problem formulations and is broken into three subsections: shape generation, numerical procedures, and aerodynamic calculations. These go into detail on the algorithms and calculations that are performed in the background of the software to provide the user with the desired results. Following the explanation of the methods used is the main results section. This is split into four sections: the graphical user interface, contour plots, streamline plots, and the simulation performance. The report then finishes with the conclusion and future work ideas for how this project can be improved.

2 Method

2.1 Shape Generation

The user has 3 different methods for inputting shapes into the CFD simulation software: 1) importing files from the internet, 2) plotting points onto the figure that is provided, or 3) freehand drawing a shape using their mouse. Providing these 3 different methods give the user the multiple ways to generate their shape of interest depending on if they need to use a predefined shape, a rough sketch, or a detailed drawing. A detailed description of each shape generation method are provided in the following subsections.

2.1.1 Importing Files

The first option available to the user is to import predefined shapes from the internet. It is suggested that the user downloads *.txt or *.dat files from the Airfoil Tools website [7]. This website provides lots of different airfoils that are used in the aerospace industry. Providing this to the user gives them a baseline to compare their more innovative testing to as the data files on [7] provide airfoils with all different chord lengths, cambers, and cross sections.

To implement the first method, the application uses the matplotlib library. It's designed around the formatting of the data files provided on the Airfoil Tools website [7]. When providing a file with similar formatting, it will properly filter out unnecessary information, place all the data points into an array, and plot out the shape provided in the file. Here is an example of how a NACA 12 airfoil is displayed:

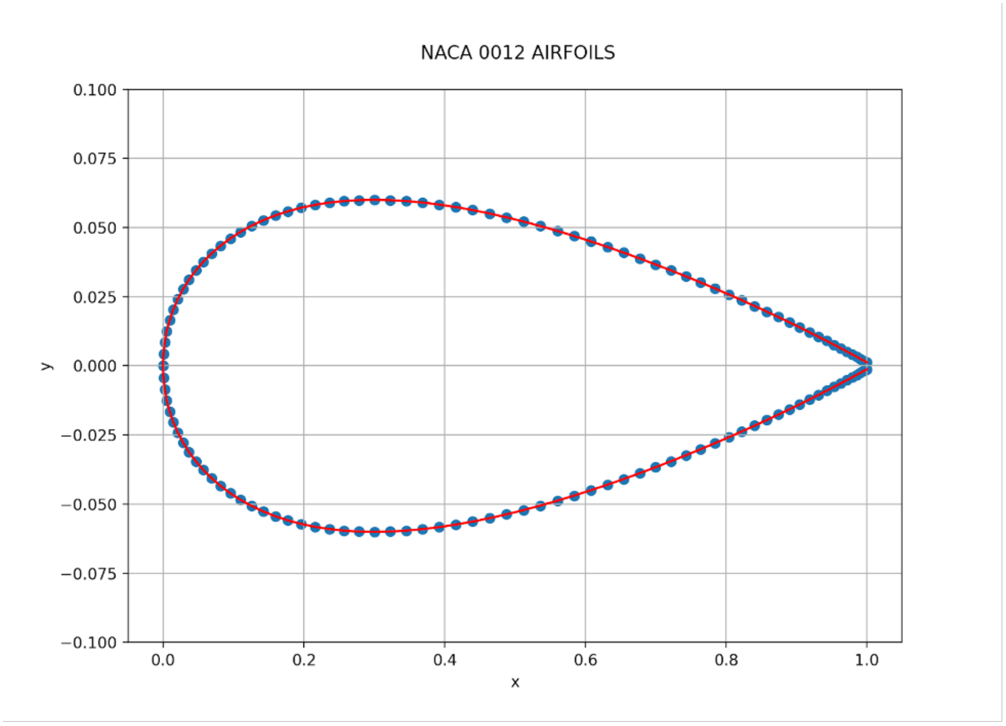


Figure 1: Imported NACA 0012 Airfoil

There are multiple steps for processing the files provided from the Airfoil Tools website [7] and creating outputs for the rest of the application to use. The files from [7] are formatted in the following manner:

- First line is the title of the file (eg. NACA 0012 Airfoil)
- Second and third lines are either empty or irrelevant
- Fourth line through the nth line contain the x,y coordinates for the top half of the shape
- n+1 line is empty
- n+2 line to the end of the file are the bottom half of the surface starting at the same point in the fourth line

The variable n can change depending on how many points make up the top half of the shape. The function AirfoilPlotter.py was created to properly read in these files and format it to meet the needs of other functions in the application. AirfoilPlotter.py has knowledge of the formatting characteristics shown in the previous bulleted list, and it uses this knowledge to create several different lists. When reading in the text file, it assigns the title of the file to a variable and creates two lists of x,y coordinates (one for the top of the shape and another for the bottom of the shape). Once these two lists are formed, the function then reverses the other of the bottom half coordinates and then reconnects the top and bottom coordinate lists. This step is important because numerical schemes require that the list of points traces the shape in the clockwise direction starting at the trailing edge. After rebuilding the coordinate lists, the function then plots the shape using the Matplotlib library and presents it to the user. This algorithm is summarized below:

Algorithm 1: Importing Shape Files

```
Result: Shape coordinates saved in Airfoil.txt file
Initialize the output coordinate lists (top and bottom) to empty;
Read the input file selected by the user;
for line in input file do
    if First line then
        Save the first line as the title;
    else if There's an empty line past the fourth line then
        Set indicator that the top half of the shape is complete;
    else if Fourth line and greater then
        if If top half is complete then
            Build bottom half coordinate list;
        else
            Build top half coordinate list;
end
Reverse the order of the bottom half points;
Connect top half and bottom half lists to form 1 list of all points in the clockwise direction
  starting at the trailing edge;
Output the list of points to Airfoil.txt;
Plot the shape using Matplotlib;
```

As mentioned above, this method is an important one for our user to take advantage of since it provides them with a performance baseline for them to compare their more innovative ideas to. The next two sections go into more detail on the methods the user has for experimenting with creative shape designs.

2.1.2 Point-by-Point

The first method for creating a new geometry lets the user to draw point-by-point. This method is based on a Matplotlib canvas embedded in the main window of the CFD GUI. This method allows the user to carefully design, modify, and prototype novel shapes with a lot of detail. Mouse-event handler functions allow the user to draw and erase new points.

The user draws new nodes by left-clicking on to the canvas. A green circle indicates the starting point of the geometry, while the rest of the captured clicks are shown in blue. As the user hovers their mouse, an orange marker a dashed line show how would the geometry change if the user decides to click. Right-clicking on the figure erases the last node drawn on the geometry.

At each left-click, the distance between the first and the last point is computed to determine if the user has closed the geometry. If the distance between the first and last click falls below a threshold of 0.02, the geometry automatically closes and drawing stops; this is indicated with a red marker.

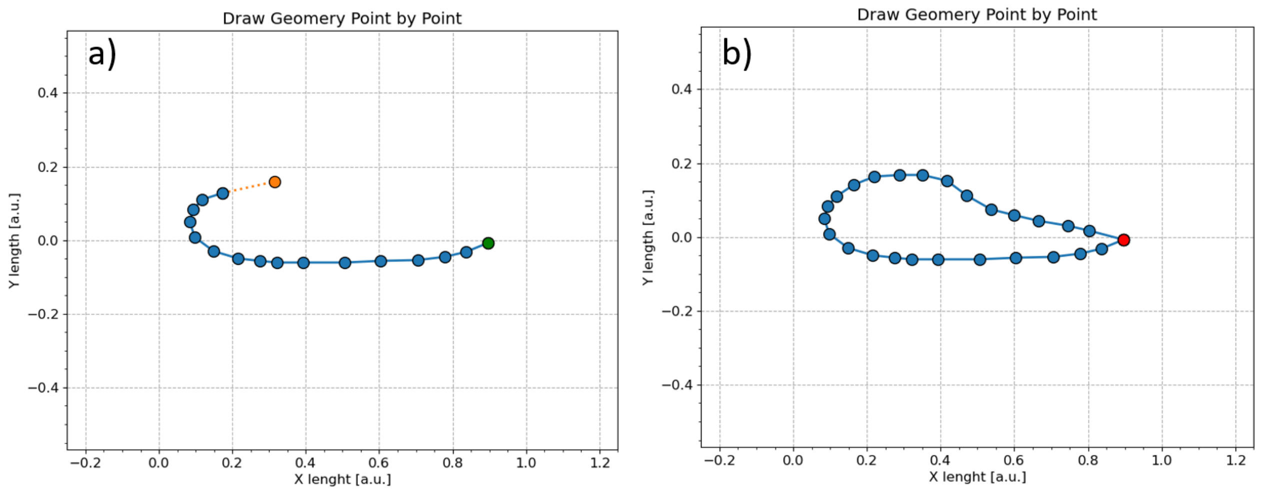


Figure 2: a) Point-by-point geometry input. b) Closed point-by-point geometry.

2.1.3 Freehand Drawing

The final shape generation method provided by the CFD application is a Freehand Drawing interface. This provides a flexible and precise option for the user to precisely draw shapes of interest using their mouse. To enable the user the capability of freehand drawing shapes, this section of the project uses the Turtle library in Python. Turtle provides the user with a virtual canvas for the user to sketch their designs on. The rules the user must follow when using this method are as follows:

- The shape must be drawn in a clockwise direction
- The shape outline can include no more than 1 intersection

The first rule stems from the need to have the list of points trace the shape in the clockwise direction, this is a requirement of the numerical procedures for generating the CFD plots. The second rule is there because the CFD application would have difficulty determining what the shape is if the line has multiple overlapping points. When the user follows this rule, the tool has the capability of either connecting the endpoints if there is no intersection or shaving off any access points that are before or after the intersection point. The process for freehand drawing is to select the freehand draw option on the GUI, then draw the shape of interest while following the two rules, and when complete the user needs to close the turtle canvas so the CFD application can process their drawing. Their drawing is then converted from the Turtle canvas into a text file and plotted on a Matplotlib figure for the rest of the function to utilize. An example of the process is shown in the figures below:

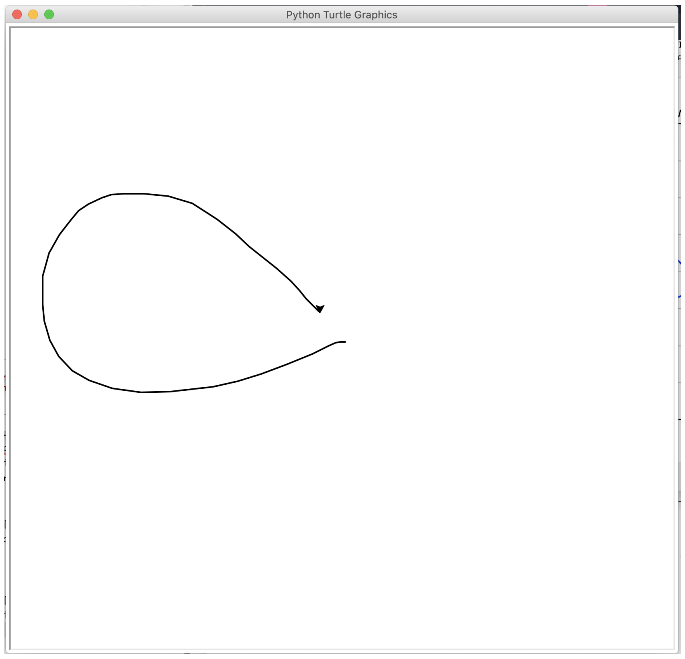


Figure 3: Freehand Drawn Shape in Turtle

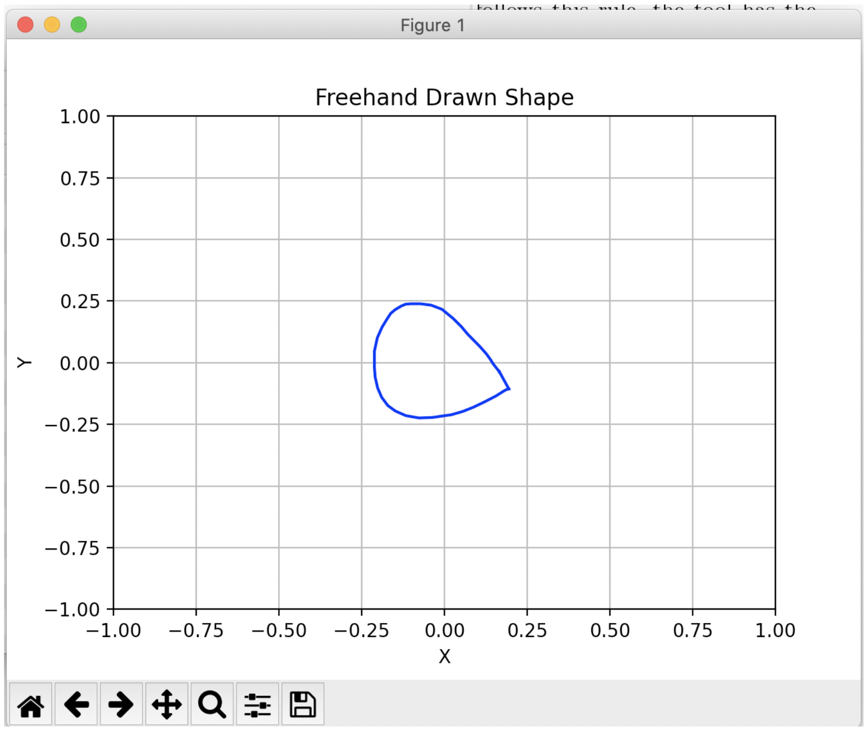


Figure 4: Converting the Turtle Shape to Matplotlib

The function that is responsible for taking the user input of the freehand drawn shape and processing it into usable data for the rest of the application is the FreehandDraw.py function. To begin, the function sets up the canvas and pen for the user to draw with. Then the user can drag the Turtle cursor across the canvas to create the shape they want to run a simulation with. While the user is dragging their mouse and drawing the shape, the FreehandDraw.py function is storing the x and y coordinates of the mouse into two arrays to build a list of the shape coordinates. Once the user is finished with their drawing and close the Turtle window, the FreehandDraw.py script then begin processing the drawing into a usable set of data. A downside to using the Turtle library here is that the x and y coordinates of the drawing are in pixels and need to be converted to match the units of the other generated data sets.

To process the Turtle drawing, the first step is to translate the center of the object to the origin. This is accomplished by calculating the average of all points in the arrays (first for x, then for y). Once the average is found, the shape is translated by subtracting the average from each element of the arrays:

$$\vec{x} = \vec{x} - avg(\vec{x}), \quad \vec{y} = \vec{y} - avg(\vec{y})$$

After translating the object, the script then down scales the size of the object to fit within a -1 to 1 x,y window. To perform this action, the unit vectors for x and y are calculated using the following equations:

$$\hat{x} = \frac{\vec{x}}{||\vec{x}||}, \quad \hat{y} = \frac{\vec{y}}{||\vec{y}||}$$

Once the shape coordinates are converted to unit vectors, the shape can then be scaled to match the desired units. The final step before finalizing the list of coordinates is to check if the drawn line has an intersection point. It is preferred to have the user leave their drawing slightly open ended and have the script connect the endpoints, but if there's an intersection point the script can provide protection for that. To find an intersection point, the function searches through the coordinate list to find any two line segments that intersect within the list. For each line segment, it calculates the line equation C=Ax+By via the following equations:

$$A = y(i+1) - y(i)$$

$$B = x(i) - x(i+1)$$

$$C = Ax(i) + By(i)$$

After obtaining the equation of each line, it then calculates the coordinates of intersection by:

$$x = \frac{B2 * C1 - B1 * C2}{A1 * B2 - A2 * B1}$$

$$y = \frac{A1 * C2 - A2 * C1}{A1 * B2 - A2 * B1}$$

where 1 and 2 denote the coefficients of line segment 1 and 2

There is a divide by zero protection since that scenario is present if the two lines being compared are parallel. This determines if the lines intersect, to then check if the line segments intersect the function ensure that the intersection point is in the domain of the two line segments. If it is in the domain, then all values before the point of intersection and after the point of intersection are removed to eliminate the excess coordinates. This not only ensures that the shape is well defined, but also prevents errors form occurring in the downstream calculations. A summarized version of the freehand drawing algorithm is below:

```
Algorithm 2: Freehand Drawing Shapes
Result: Shape coordinates saved in FreehandDraw.txt file
Setup the Turtle canvas and pen for the user to draw with;
while The user is drawing the shape do
    Build x and y arrays that contain the mouse coordinates;
end
Translate the center of the shape to the origin;
Calculate the unit vectors of x and y;
Scale the shape;
for i in range(2 to length(x)/2) do
    for k in range(length(x)/2 to length(x)) do
        Calculate equation of the line containing points (i-1, i);
        Calculate equation of the line containing points (k-1, k);
        Find point of intersection between the two lines;
        if Intersection point is in line segment domain then
            Break both loops;
    end
end
if An intersection was found then
    Remove excess points before and after intersection;
Output the list of points to FreehandDraw.txt;
Plot the shape using Matplotlib;
```

This method of shape generation provides an efficient way for the user to sketch and test their ideas. This is the most flexible future to take advantage of if the shape of interest is abnormal and requires a more creative design to generate.

2.2 Numerical Procedures: Source and Vortex Panel Methods

We implement a combination of source panel method (SPM) and vortex panel method (VPM) to evaluate the flow around the object and measure aerodynamic properties. In this method, we discretize the surface of the object into N panels, as shown in Figure-(5) (a) (in clockwise direction). Figure-(5) (b) depicts the geometric properties on each panel.

- (x_i, y_i) represents the start point and (x_{i+1}, y_{i+1}) represents the end point of panel i.
- (x_c, y_c) is the mid-point coordinate of the panel.
- S is the length of the panel
- n is the outward pointing normal to the panel
- ϕ is the angle from the positive x to the inside of the panel.
- δ is the angle from the positive x to the outward pointing normal on the panel.
- β is the angle between the free-stream velocity and normal.

In the next two sections (Sec-2.2.1 and Sec-2.2.2) we describe how normal and tangential component of velocity is calculated over a panel using source panel and vortex panel method.

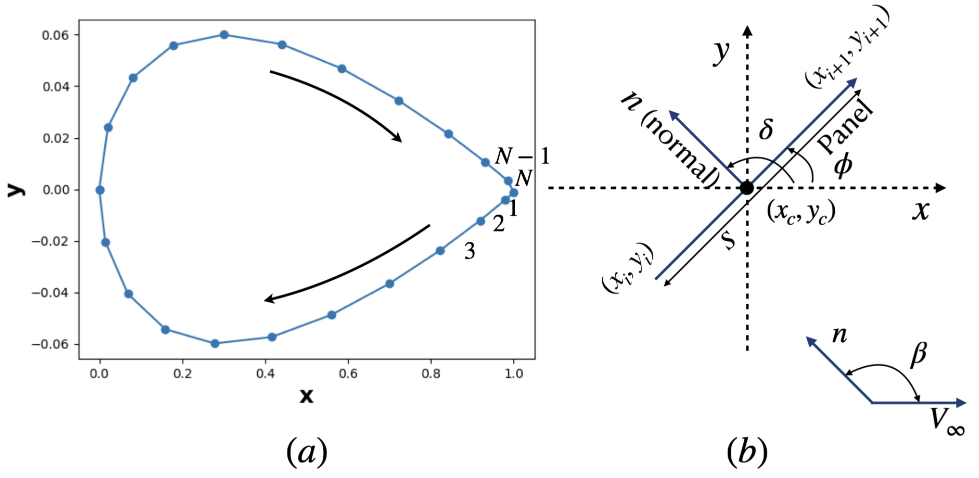


Figure 5: The surface of the object is discretized into panels

2.2.1 Normal and Tangential Velocities for Source Panel Method

The normal velocity on a panel, using source panel method, is given by equation:

$$V_{ni} = V_\infty cos(\alpha)\frac{dx_i}{dn_i} + V_\infty sin(\alpha)\frac{dy_i}{dn_i} + \sum_{j=1}^{N}\frac{\lambda_j}{2\pi}\int_j \frac{d}{dn_i}ln(r_{ij})dS_j$$

The subscript n represents the normal velocity and the subscripts i and j give the position of the panel, i and the position on the panel, j. The first two terms in the equation are the calculation of the freestream velocity in the x and y direction on each panel where α is the angle of attack. The last term in the equation is the source normal velocity induced at each panel. The integral over j is the integral over the length of each panel where r_{ij} is the distance from the point j on the panel to the center point of the panel.

In order to make these equations more accessible to code, [6] took further steps to simplify the above equation. The first simplification can be is done on the first two velocity terms and comes out to be:

$$V_\infty cos(\alpha)\frac{dx_i}{dn_i} + V_\infty sin(\alpha)\frac{dy_i}{dn_i} = V_\infty cos(\beta)$$

This simplification gets rid of the problem of taking derivatives and allows the user to input a value, β' calculated when breaking up the geometry. This value is the same for all of the normal and tangential velocities in both the vortex and source panel methods.

While the first two terms of the velocity equations can be easily simplified and reused for both the normal and tangential velocities of both the vortex and source panel methods the last term in each equation needs independent simplification, Thus, following equation for the integral for the normal velocity in the source panel method was derived by [6] as:

$$I_{ij} = \frac{C}{2}ln(\frac{S_j^2 + 2*AS_j + B}{B}) + \frac{D - AC}{E}[\arctan(\frac{S_j + A}{E}) - \arctan(\frac{A}{E})]$$

Where

$$A = -(x_i - x_j)cos(\psi_j) - (y_i - y_j)sin(\psi_j)$$

$$B = (x_i - x_j)^2 + (y_i - y_j)^2$$
$$C = sin(\psi_i - \psi_j)$$
$$D = -(x_i - x_j)sin(\psi_j) - (y_i - y_j)cos(\psi_j)$$
$$E = \sqrt{B - A^2}$$

Where I_{ij} is the integral in the last part of the normal velocity equation given above, S_j is the length of the jth panel, x_i and y_i are the center points of each panel, x_j and y_j are the starting points of each panel, and ψ_i and ψ_j are the orientation of the jth positon on the ith panel. [6]

These values were calculated in our code when breaking up the geometry that was input and thus could be input into the code used from [6].

The tangential velocity also needed to be taken at each panel and the equation for that is similar to the equation for the normal velocity with slight changes.

$$V_{ti} = V_\infty cos(\alpha)\frac{dx_i}{dt_i} + V_\infty sin(\alpha)\frac{dy_i}{dt_i} + \sum_{j=1}^{N}\frac{\lambda_j}{2\pi}\int_j \frac{d}{dt_i} ln(r_{ij})dS_j$$

The above equation is the equation for the tangential velocity at each panel for the Source panel method. The only difference in this equation and the normal velocity equation is the derivatives are taken with respect to the tangent instead of the normal at each panel. For the tangential velocity, the last term in the equation was once again broken up into a term that was easier to code into python by [6]. The equation for that term is as follows.

$$J_{ij} = \frac{C}{2}ln(\frac{S_j^2 + 2*AS_j + B}{B}) + \frac{D - AC}{E}[\arctan(\frac{S_j + A}{E}) - \arctan(\frac{A}{E})]$$

Where C and D are changed to account for the tangential as opposed to the normal.

$$A = -(x_i - x_j)cos(\psi_j) - (y_i - y_j)sin(\psi_j)$$
$$B = (x_i - x_j)^2 + (y_i - y_j)^2$$
$$C = cos(\psi_i - \psi_j)$$
$$D = (x_i - x_j)cos(\psi_j) - (y_i - y_j)sin(\psi_j)$$
$$E = \sqrt{B - A^2}$$

All values in the equation are the same as for the normal velocity, and once again the values that were calculated from our geometry code were able to be implemented into these equations in python.

2.2.2 Normal and Tangential Velocities for Vortex Panel Method

The vortex panel method was used along with the source panel method to create a more precise measurement of necessary aerodynamic values. The equations for the normal and tangential velocities of the vortex panel method are similar to the equations derived for the source panel method and use the same input values, making the implementation of the code for our uses easier.

The equation for the vortex method normal velocity is given as

$$V_{ni} = V_\infty cos(\alpha)\frac{dx_i}{dn_i} + V_\infty sin(\alpha)\frac{dy_i}{dn_i} + \sum_{j=1}^{N}\frac{\gamma_j}{2\pi}\int_j \frac{d(\theta_{ij})}{dn_i}dS_j$$

Once again, the integral part of the above equation was further simplified to ease its integration into python. The simplification of that portion of the equation by [6] is given as.

$$K_{ij} = \frac{C}{2} ln(\frac{S_j^2 + 2 * AS_j + B}{B}) + \frac{D - AC}{E}[\arctan(\frac{S_j + A}{E}) - \arctan(\frac{A}{E})]$$

Where

$$A = -(x_i - x_j)cos(\psi_j) - (y_i - y_j)sin(\psi_j)$$
$$B = (x_i - x_j)^2 + (y_i - y_j)^2$$
$$C = -cos(\psi_i - \psi_j)$$
$$D = (x_i - x_j)cos(\psi_j) + (y_i - y_j)sin(\psi_j)$$
$$E = \sqrt{B - A^2}$$

The equation for the vortex method tangential velocity is given as

$$V_{ti} = V_\infty cos(\alpha)\frac{dx_i}{dn_i} + V_\infty sin(\alpha)\frac{dy_i}{dn_i} + \sum_{j=1}^{N} \frac{\gamma_j}{2\pi} \int_j \frac{d(\theta_{ij})}{dt_i} dS_j$$

And simplifying the integral as before gives

$$L_{ij} = \frac{C}{2} ln(\frac{S_j^2 + 2 * AS_j + B}{B}) + \frac{D - AC}{E}[\arctan(\frac{S_j + A}{E}) - \arctan(\frac{A}{E})]$$

Where

$$A = -(x_i - x_j)cos(\psi_j) - (y_i - y_j)sin(\psi_j)$$
$$B = (x_i - x_j)^2 + (y_i - y_j)^2$$
$$C = sin(\psi_i - \psi_j)$$
$$D = (x_i - x_j)sin(\psi_j) - (y_i - y_j)cos(\psi_j)$$
$$E = \sqrt{B - A^2}$$

2.2.3 Linear system of equation

Using the combined source and vortex panel method described above, the normal component of velocity on a panel i is given by:

$$V_{n,i} = V_\infty \cos\beta_i + \sum_{j=1}^{N} \frac{\lambda_j}{2\pi} \int_j \frac{d}{dn_i} ln(r_{ij}) dS_j - \sum_{j=1}^{N} \frac{\gamma_j}{2\pi} \int_j \frac{d(\theta_{ij})}{dn_i} dS_j \tag{1}$$

In the above equation, the first term is the contribution from freestream velocity, second term is the contribution from source panel method and thirst term is from vortex panel method. The total number of unknowns in the above equation are N+1: source strength λ_i on N panels and vortex strength γ (vortex strength is same on each panel). The N equations are obtained by applying the flow tangency boundary condition i.e., $V_{ni} = 0$ on each panels (as flow cannot pass through the solid object). Upon applying this boundary condition, Eq-1 simplifies to:

$$V_\infty \cos\beta_i + \sum_{\substack{j=0 \\ j\neq i}}^{N} \frac{\lambda_j\, I_{ij}}{2\pi} + \frac{\gamma}{2} - \sum_{\substack{j=0 \\ j\neq i}}^{N} \frac{\gamma\, K_{ij}}{2\pi} = 0 \tag{2}$$

Eq-2 applied to the N panels provides the N equations. The last equation to solve the $N+1$ unknowns is obtained from Kutta-condition by setting the first and last panel velocity equal to each other:

$$V_{t,N} = -V_{t,1} \tag{3}$$

where,

$$V_{t,1} = V_\infty \sin\beta_1 + \sum_{j=2}^{N} \frac{\lambda_j\, J_{1j}}{2\pi} + \frac{\gamma}{2} - \sum_{j=2}^{N} \frac{\gamma\, L_{1j}}{2\pi}$$

and

$$V_{t,N} = V_\infty \sin\beta_N + \sum_{j=1}^{N-1} \frac{\lambda_j\, J_{Nj}}{2\pi} + \frac{\gamma}{2} - \sum_{j=1}^{N-1} \frac{\gamma\, L_{Nj}}{2\pi}$$

The computation of integrals I and J from source panel method and K and L from vortex panel method is present in file *functions.py* under function definitions *COMPUTE_SOURCE_PANEL* and *COMPUTE_VORTEX_PANEL*. From the above $N+1$ equations we build the matrix system:

$$A\,X = B$$

where the first N rows of column X are source strengths λ_i and the last row is γ.

2.2.4 Velocity at arbitrary point P

Once the source and vortex strengths are evaluated, we can compute velocity at any arbitrary point P $(V_{x,P}, V_{y,P})$ exterior to the object using the equations given below.

$$V_{x,P} = V_\infty cos(\alpha) + \sum_{j=2}^{N} \frac{\lambda_j\, Mx_{Pj}}{2\pi} - \sum_{j=2}^{N} \frac{\gamma\, Nx_{Pj}}{2\pi} \tag{4}$$

$$V_{y,P} = V_\infty sin(\alpha) + \sum_{j=2}^{N} \frac{\lambda_j\, My_{Pj}}{2\pi} - \sum_{j=2}^{N} \frac{\gamma\, Ny_{Pj}}{2\pi} \tag{5}$$

The computation of integrals Mx and My from source panel method and Nx and Ny from vortex panel method is present in file *functions.py* under function definitions *STREAMLINE_SPM* and *STREAMLINE_VPM*.

2.3 Aerodynamic Calculations

The aerodynamic coefficients of interest are pressure, lift, drag, and moment. From the panel methods, the coefficient or pressure on each panel can be calculated as function of velocity [8]:

$$C_p = 1 - (\frac{u_t}{v_\infty})^2$$

where u_t is the tangential velocity on a panel and v_∞ is the freestream velocity. To calculate the lift coefficient, project the contribution of the pressure coefficient on each panel to the x-axis, divide by the chord length, and sum all of the panels together. The projection can be done by simply finding the difference in the x values of the start and end points of each panel:

$$C_l = \sum_{n=1}^{N} C_p \frac{x_{n+1} - x_n}{c}$$

where N is the total number of panels, c is the chord length, and $x_{n+1} - x_n$ is the difference in the x-values of the end points on each panel. The coefficient of drag can be calculated in a similar manner, replacing x with y for each panel.

$$C_d = \sum_{n=1}^{N} C_p \frac{y_{n+1} - y_n}{h}$$

However, for the case of inviscid, incompressible, and irrotational flow, the drag should be equal to zero. This attribute can be observed by increasing the number of panels. As the resolution increases, the drag coefficient will approach zero, where the lift and moment coefficients will remain about the same. To calculate the moment coefficient about the quarter-chord, first define the location of the quarter-chord as one-fourth of the length of the chord from the leading edge on the x-axis:

$$c_{1/4} = \frac{max(x) - min(x)}{4} + min(x)$$

This gives the x location of the quarter-chord. Multiply the contribution of the pressure coefficient on each panel by the distance from the center of each panel to the quarter-chord location to get the contribution to the moment from lift. The contribution to moment from drag is negligible, but is included in a similar fashion, where the contribution from each panel is multiplied by the height of the center of each panel. As the calculation moves around the airfoil, the signs remain consistent with the right-hand rule pointing into page, where clockwise is positive if the calculation first travels over the top surface and ends on the bottom surface:

$$C_m = \sum_{n=1}^{N} C_p (\frac{x_{n+1} + x_n}{2} - c_{1/4}) \frac{x_{n+1} - x_n}{c} + \sum_{n=1}^{N} C_p (\frac{y_{n+1} + y_n}{2}) \frac{y_{n+1} - y_n}{h}$$

For preliminary design, the lift and moment coefficients that were calculated are suitable as a rough estimate for performance, but should not be used for any sort of final design. To determine actual values for lift and moment from the coefficients, multiply by the following user-input constants:

$$L = \frac{1}{2} \rho v_\infty^2 A C_l$$

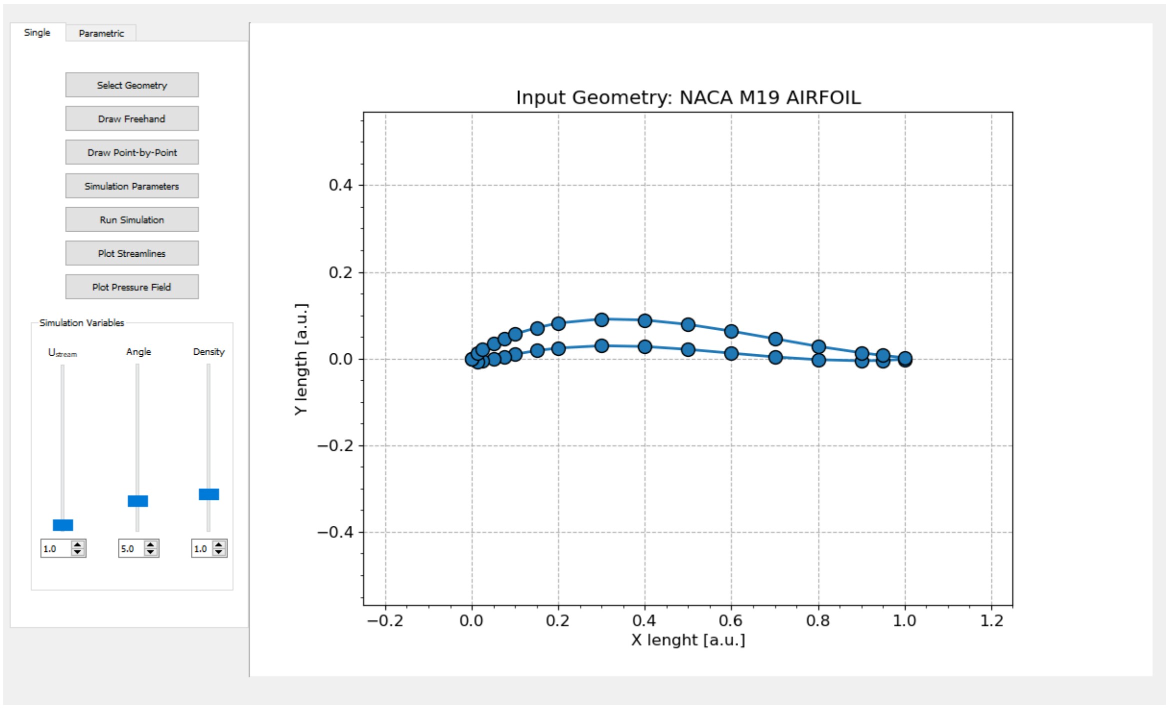
$$M = \frac{1}{2} \rho v_\infty^2 A C_m$$

where A is the wing area. The purpose of the code is for 2-D flow for educational purposes, so the wing area is not included as an input to determine the actual lift or moment values.

3 Main Results

3.1 Graphical User Interface

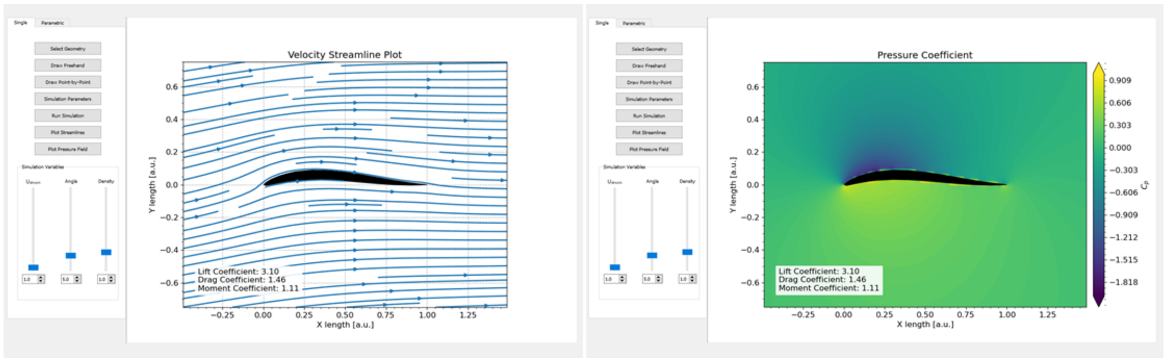
To improve the interactivity and power of the CFD tool, a graphic user interface (GUI) was developed. The tool integrates the user input, geometry design, plotting, simulation, and exploration of the results. The GUI was developed using PyQt and Qt Designer. The interface allows the user to select .txt files with different airfoil geometries, or draw their own using the freehand or point-by-point method. The selected geometry is then displayed on the main window.



U_stream: 1.0
Angle: 5.0
Density: 1.0

Figure 6: Main window of the CFD graphic user interface in single simulation mode.

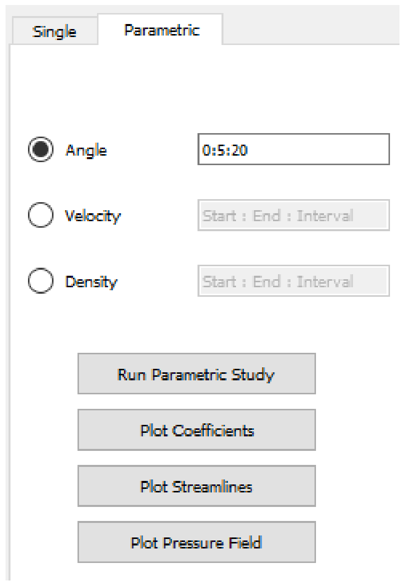
The tool can be used in two modes: 1) single flow simulation at specific conditions, and 2) parametric simulation study at a range of specified conditions. In the single simulation, the user can specify the specific upstream velocity, the angle of attack, and the density of the fluid. Once the inputs are completed, the user can initiate the simulation by hitting the "Run Simulation" button. The progress of the simulation is displayed on the console. A pop-up window notifies the user that the simulation is complete; this enables the exploration of the results under "Plot Streamlines" and "Plot Pressure Field". The plots also show the lift, drag, and moment coefficients calculated for the geometry at the specified condition.



Lift Coefficient: 3.10
Drag Coefficient: 1.46
Moment Coefficient: 1.11

Figure 7: Main window showing the velocity streamlines and pressure field.

In the second mode, the user can specify a range of angles, velocities, or densities to use for the simulation. This mode offers the opportunity to explore how the flow changes as function of the different parameters in order to better understand and discover trends.



(●) Angle 0:5:20
( ) Velocity
( ) Density

Figure 8: Options for performing a parametric study.

Once the simulation is complete, the user can explore the results in different ways. The interface gives the option to plot the lift, drag, and moment coefficient as a function of the given parameter. The user can also explore the pressure field and velocity streamlines for the simulated conditions. Keyboard event handlers in Matplotlib were implemented to enable easy data exploration for the pressure and velocity results. The user can advance in the pressure field by pressing 'a' and 'd' on their keyboard.

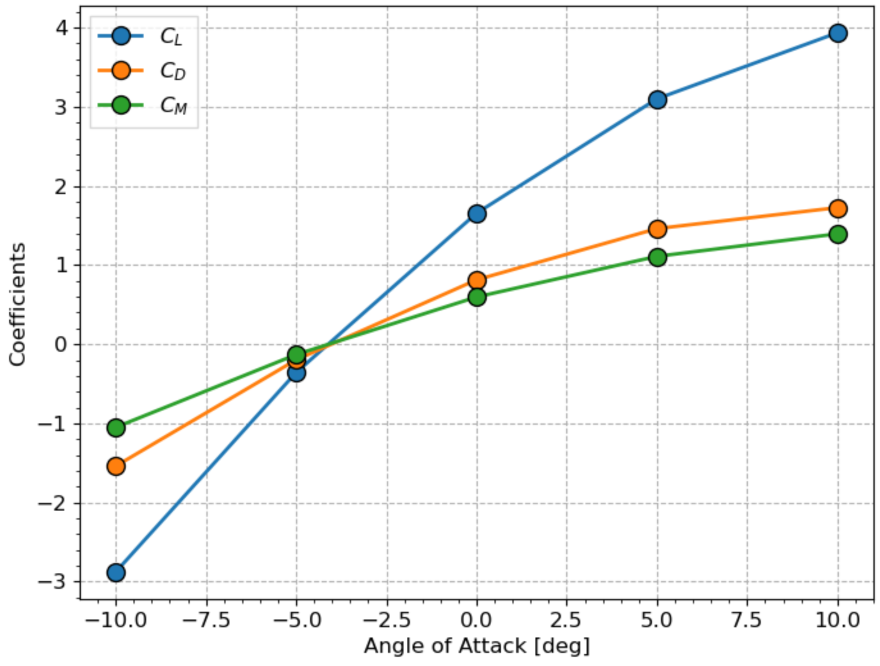


Figure 9: Sample lift, drag, and moment coefficients for a parametric study on the effects of the angle of attack.

3.2 Contour Plot

The contour plot was created using simple coefficient of pressure calculation:

$$C_p = 1 - (\frac{u_t}{v_\infty})^2$$

u_t was determined by taking the velocity at each point in the mesh V_x, V_y and taking the magnitude. The v term is input by the user on the GUI. The plot was created using an object oriented approach to better integrate it into the GUI.

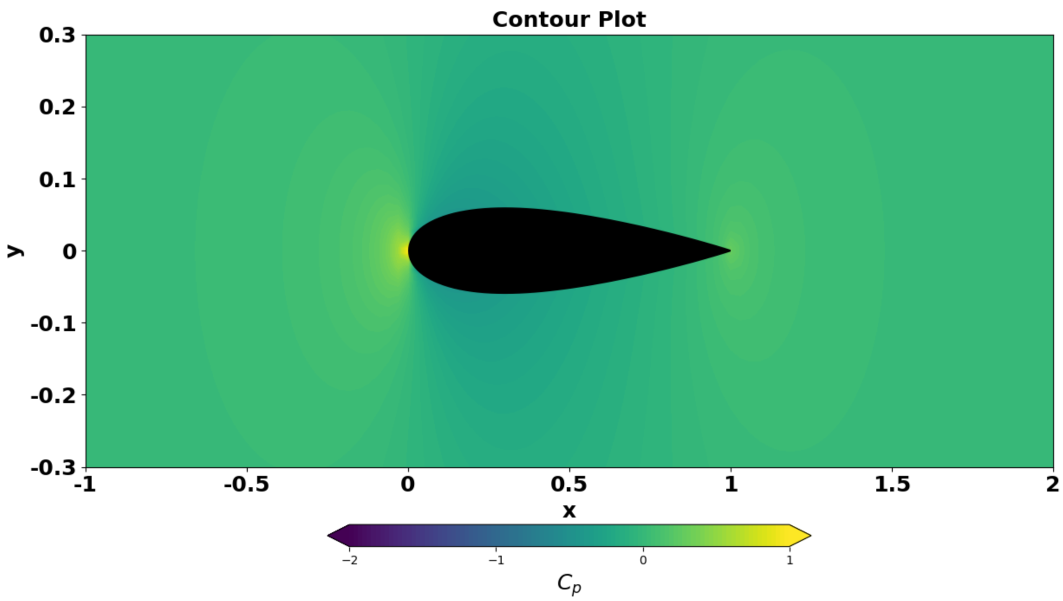


Figure 10: Pressure coefficient variation around the NACA 0012 airfoil.

This contour plot was plotted by calculating the coefficient of pressure at every point on the grid and plotting it, then taking the shape of the airfoil and filling it in. Figure-(10) shows the variation of coefficient of pressure over the grid for NACA 0012 airfoil.

3.3 Streamline Plot

In order to plot the streamline-pattern around the object we need the information of x and y components of velocity over the grid. The velocity at any arbitrary point P can be represented as a combination of contributions from freestream velocity, vortex panels, and source panels that approximate the geometry (Eq-4 and Eq-5). We use the *streamline()* function from the *matplotlib* library to draw the vector flow field around the object. Figure-(11) shows the 2-D streamline-pattern around the NACA 0012 airfoil.

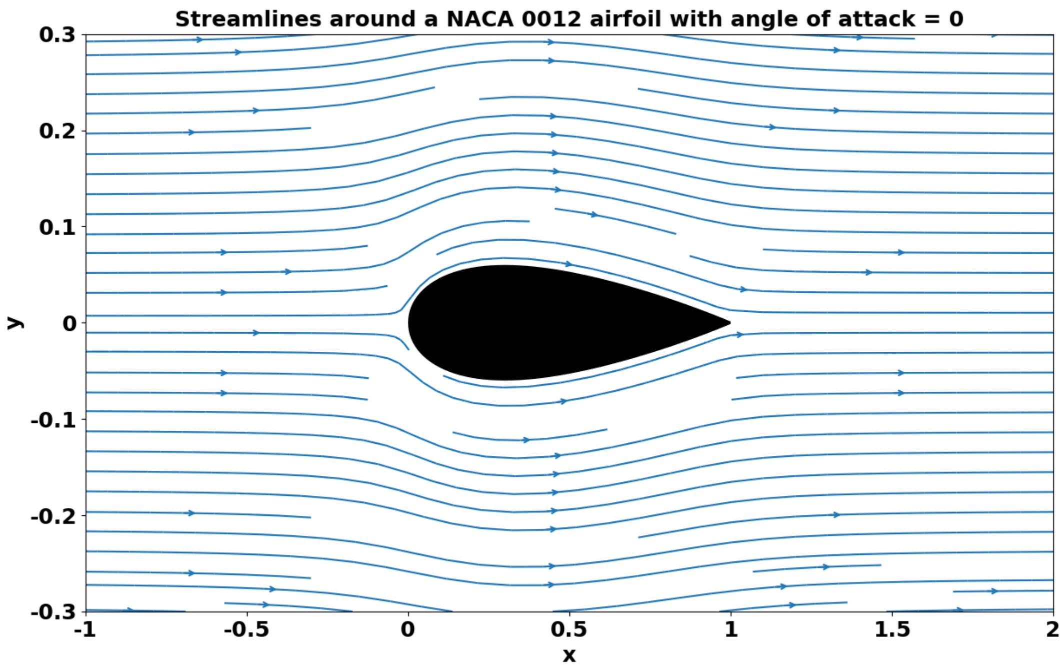


Figure 11: Streamline pattern around the NACA 0012 airfoil

3.4 Simulation Performance

We compare the results generated from our numerical procedure with the results from [5]. The numerical procedure of [5] involves the computation using the source panel method alone. Here we implemented a combination of source and vortex panel method to have a more robust performance. As can be noted from Figure-(12) and Figure-(13), the streamline and contour plots matches well with the results from [5].

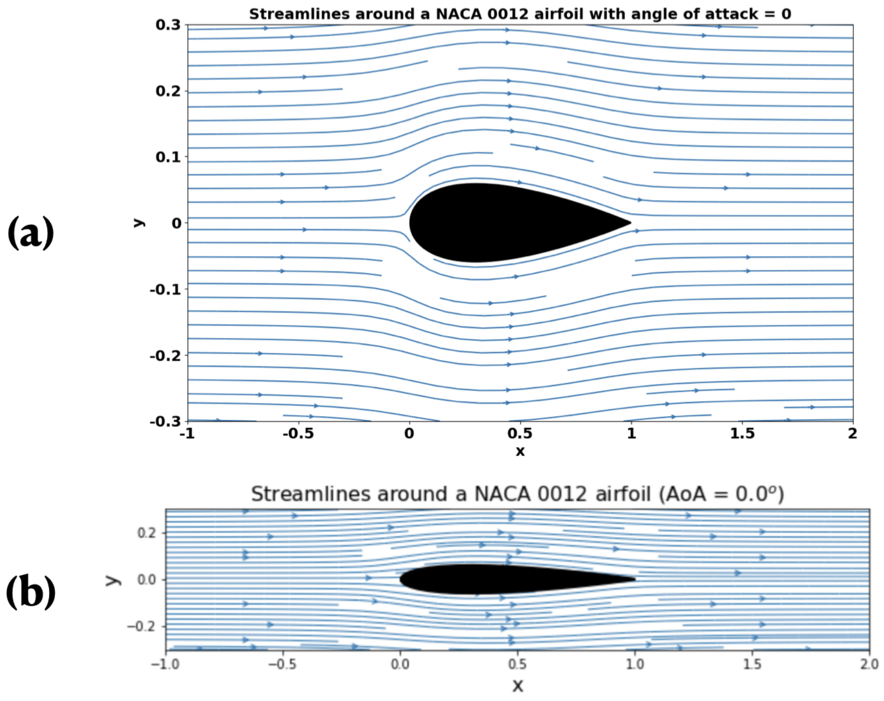


Figure 12: Compares the streamline-pattern around the NACA 0012 airfoil from (*a*) our numerical procedure and (*b*) Aeropython [5].

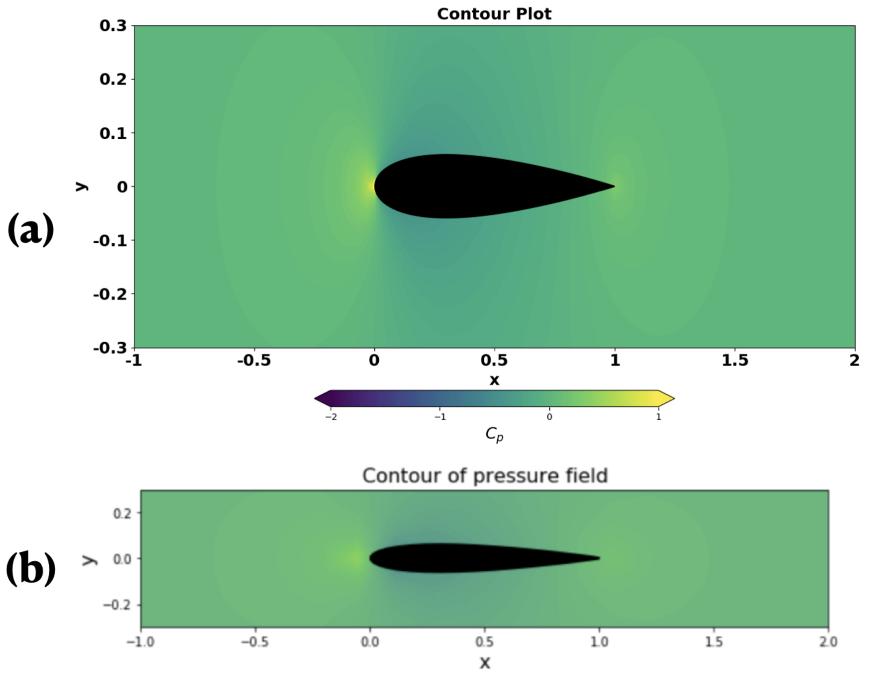


Figure 13: Compares the contour of pressure field around the NACA 0012 airfoil from (*a*) our numerical procedure and (*b*) Aeropython [5].

4 Conclusion and Future Work

In conclusion, the CFD user interface provides an education foundation for student to perform inviscid flow simulations with both predefined shapes and manually-created shapes. Furthermore, the simulation results from the developed tool are similar to the results seen in CFD simulations created by other researchers.

The implementation of the solver with a GUI component significantly improves the usability and impact of the tool. In addition to accepting many different shapes, the GUI has several aerodynamic parameters that can be adjusted along with a user friendly method of comparing results. The tool also includes the option to perform parametric sweeps on the different parameters. This particular feature would enable student to better understand and grasp the effect that each of the variables have on the behavior of the flow. Overall, the CFD developed in this project gives students of aerodynamics and fluid mechanics the possibility of directly engaging with the topics of inviscid flow, lift, drag, and external flow in order to further expand their knowledge of flow phenomena.

There are a few opportunities that a future researcher can explore and improve upon with this CFD user interface. One of these areas of improvement would be usability testing to improve the overall user experience and inputs, as well as additional error and exception handling. In terms of generating shapes, the drawing feature can be better integrated with the numerical procedures to calculate the flow over the shape. At the moment it has several assumptions that the user must be aware of when using the feature such as starting from the trailing edge and drawing in a clockwise direction.

References

[1] I. Currie. Fundamental mechanics of fluids: Fourth edition. *Taylor Francis Group, Bosa Roca*, 2016.

[2] N. Battista. Suite-cfd: An array of fluid solvers written in matlab and python. *Fluids*, pages 1–58, 2020.

[3] V. Garcia-Garrido M. Fontelos and U. Kindelan. Evolution and breakup of viscous rotating drops. *SIAM J. Appl. Math*, pages 1941–1964.

[4] A. Nir S. Malik, M. Lavrenteva. Shapes and viscous rotating drops in a compressional/extensional flow. *Phys. Rev. Fluids*, 2020.

[5] L. Barba. Cfdpython. *Jupyter BarbaGroup*, 2018.

[6] JoshtheEngineer. Panel methods. *https://www.joshtheengineer.com/panel-methods/*, 2020.

[7] *Airfoil Tools*, 2020.

[8] Auld and Srinivas. 2d panel methods. *Aerodynamics for Students*, 2020.

A First Appendix

Name	Modules Created	Contributions
Stephen Hannah	AirfoilPlotter.py FreehandDraw.py	• Implemented a method to import files to run simulations with • Implemented a method to enable the user the freehand drawing capability • Gathered sample data sets • Scheduled meetings and outlined work
Luis G. Gutierrez	main_gui.py CFD_gui_param.py sim_params.py CFD_simulation.py	• Designed graphic user interface in Qt Designer. • Implemented event handling functions and connections in PyQt. • Coded point-by-point drawing method. • Created parametric study feature. • Integrated all the functions into GUI application.

Christopher Zoller and Natasha Singh	Geometry.py Functions.py getvelocity.py getgrid.py plotCp.py plotVfield.py	• Worked on understanding [6] in order to break up creating velocity field • Used equations from [6] to create necessary geometric inputs from geometry • Took geometric inputs and modified code from [6] to fit our inputs • Solved linear system of equations to get velocity field for streamline and contour plots. • Converted plots of velocity field and contour and made them object oriented for the GUI
Matthew Liu	Coeffs.py	• Conducted research to find article [8] to calculate aerodynamic coefficients • Created code to calculated lift, drag, and moment coefficients from pressure coefficient

B Second Appendix

Module	Libraries Used	Purpose
AirfoilPlotter.py (Integrated into main_gui.py)	Matplotlib Numpy	Reads in an imported file with a predefined shape, plots it in Matplotlib, and processes it into data arrays for downstream calculations. See Section 2.1.1 for detailed description.
FreehandDraw.py	Matplotlib Numpy Math Turtle	Sets up a drawing canvas in Turtle for the user to create the shape they want to test. The shape is then processed, re-plotted in Matplotlib, and output into data arrays for downstream calculations. See Section 2.1.3 for detailed description.
Geometry.py (Integrated into cfd_simulation.py)	Numpy	Takes the list of x and y values from the geometry and creates lists of geometric values necessary for making panel method calculations. See section 2.2.1 for more details on the values it outputs
Functions.py	Numpy Math	Takes in the geometric values from the list to create a matrix of tangential and normal integrals for the source and vortex panel methods, taken and modified from [6] to work with our code. See all of section 2.2 for in depth description.
getvelocity.py (Integrated into CFD_utils.py)	Numpy Math Matplotlib Functions	Gets velocity values from outputs of matrix necessary to find gama and lamba values for the tangential and normal velocities. See section 2.[] for more details
getgrid.py (Integrated into CFD_utils.py)	Numpy	creates a grid the velocity field and contour plot can be plotted over.
plotCp.py (Integrated into CFD_utils.py)	Matplotlib	creates a coefficent of pressure plot in object oriented plotting. See section 3.2 for more details.
plotVfield.py (Integrated into CFD_utils.py)	Matplotlib	creates a velocity field plot in object oriented plotting. See section 3.3 for more details.
Coeffs.py (Integrated into CFD_utils.py)	Math	Uses pressure coefficients to calculate lift, drag, and moment coefficients for the 2D geometry.

main_gui.py	PyQt Numpy Os Matplotlib	Integrates all functions as to give the user a graphic interface.
CFD_utils.py	PyQt Numpy Math Matplotlib	Collection of helper functions used to perform simulation and plot results.
CFD_gui_params.py	PyQt	Main window class for GUI tool.
sim_params.py	PyQt	Dialog window class for setting simulation parameters.