Vortex Panel Method

Nicholas Richmond

Student ID: 919578115

Project #: 4

Date: 11/22/24

EAE 127: Applied Aerodynamics

University of California, Davis

In [11]:
```
#standard imports and setups
import math
import pandas as pd #type: ignore
import numpy as np #type: ignore
import os
import matplotlib.pyplot as plt #type: ignore
import matplotlib.lines as mlines
from scipy import integrate
### JUPYTER NOTEBOOK SETTINGS ########################################
#Plot all figures in full-size cells, no scroll bars
%matplotlib inline
#Disable Python Warning Output
#(NOTE: Only for production, comment out for debugging)
import warnings
warnings.filterwarnings('ignore')
### PLOTTING DEFAULTS BOILERPLATE (OPTIONAL) #########################
#SET DEFAULT FIGURE APPERANCE
import seaborn as sns #Fancy plotting package #type: ignore
#No Background fill, legend font scale, frame on legend
sns.set_theme(style='whitegrid', font_scale=1.5, rc={'legend.frameon': True}
#Mark ticks with border on all four sides (overrides 'whitegrid')
sns.set_style('ticks')
#ticks point in
sns.set_style({"xtick.direction": "in","ytick.direction": "in"})
#fix invisible marker bug
sns.set_context(rc={'lines.markeredgewidth': 0.1})
#restore default matplotlib colormap
mplcolors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd',
'#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf']
sns.set_palette(mplcolors)
```

```
#Get color cycle for manual colors
colors = sns.color_palette()
#SET MATPLOTLIB DEFAULTS
#(call after seaborn, which changes some defaults)
params = {
#FONT SIZES
'axes.labelsize' : 30, #Axis Labels
'axes.titlesize' : 30, #Title
'font.size' : 28, #Textbox
'xtick.labelsize': 22, #Axis tick labels
'ytick.labelsize': 22, #Axis tick labels
'legend.fontsize': 24, #Legend font size
'font.family' : 'serif',
'font.fantasy' : 'xkcd',
'font.sans-serif': 'Helvetica',
'font.monospace' : 'Courier',
#AXIS PROPERTIES
'axes.titlepad' : 2*6.0, #title spacing from axis
'axes.grid' : True, #grid on plot
'figure.figsize' : (8,8), #square plots
'savefig.bbox' : 'tight', #reduce whitespace in saved figures
#LEGEND PROPERTIES
'legend.framealpha' : 0.5,
'legend.fancybox' : True,
'legend.frameon' : True,
'legend.numpoints' : 1,
'legend.scatterpoints' : 1,
'legend.borderpad' : 0.1,
'legend.borderaxespad' : 0.1,
'legend.handletextpad' : 0.2,
'legend.handlelength' : 1.0,
'legend.labelspacing' : 0,
}
import matplotlib #type:ignore
matplotlib.rcParams.update(params) #update matplotlib defaults, call after
### END OF BOILERPLATE ##################################################
colors = sns.color_palette() #color cycle
```

Problem 1: Vortex Potential Flow

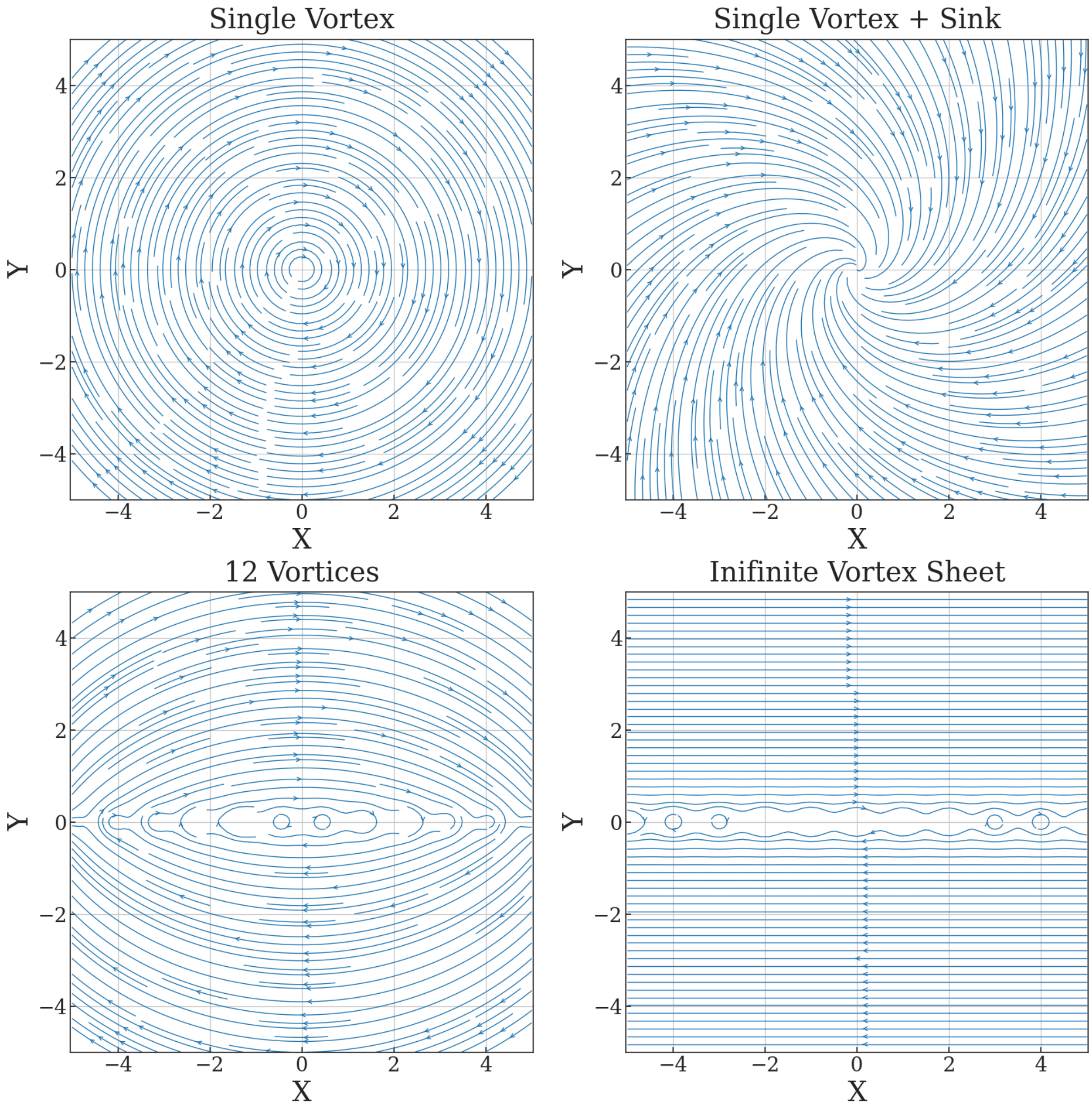
This problem asks us to plot a single vortex, 12 vorticies (finite vortex sheet), and an infinite vortex sheet (or, rather, an approximation of one).

In [12]:
```
#set up functions and meshgrid

N = 250                                          # Number of points/sections
x_start, x_end = -5.0, 5.0                       # Boundaries of our flow in
y_start, y_end = -5, 5                         # Boundaries of our flow in the

# Note, you can adjust the start and end points later to get the best image

x = np.linspace(x_start, x_end, N)               # 1D array of x points
y = np.linspace(y_start, y_end, N)               # 1D array of y points
```

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

def get_vortex_velocity(Gamma, xv, yv, X, Y):
    """
    Returns the velocity field generated by a vortex

    Parameters
    ----------
    Gamma: float
        Strength of the vortex.
    xv: float
        x-coordinate of the vortex.
    yv: float
        y-coordinate of the vortex.
    X: 2D Numpy array of floats
        x-coordinate of the mesh points.
    Y: 2D Numpy array of floats
        y-coordinate of the mesh points.

    Returns
    -------
    u: 2D Numpy array of floats
        x-component of the velocity vector field.
    v: 2D Numpy array of floats
        y-component of the velocity vector field.
    """
    # Here, input the equation for u_vortex from the equations above
    u = (Gamma/(2*np.pi))*((Y-yv)/((X-xv)**2 + (Y-yv)**2))

    # Here, input the equation for v_vortex from the equations above
    v = -1 * (Gamma/(2*np.pi)) * ((X-xv)/((X-xv)**2 + (Y-yv)**2))

    return u, v
def get_velocity(strength, xs, ys, X, Y):
    """
    Returns the velocity field generated by a source/sink.

    Parameters
    ----------
    strength: float
        Strength of the source/sink.
    xs: float
        x-coordinate of the source (or sink).
    ys: float
        y-coordinate of the source (or sink).
    X: 2D Numpy array of floats
        x-coordinate of the mesh points.
    Y: 2D Numpy array of floats
        y-coordinate of the mesh points.

    Returns
    -------
    u: 2D Numpy array of floats
        x-component of the velocity vector field.
    v: 2D Numpy array of floats
        y-component of the velocity vector field.
```

```
    """
    # Here, input the equation for u_source from the equations above
    u = strength / (2 * np.pi) * (X - xs) / ((X - xs)**2 + (Y - ys)**2)

    # Here, input the equation for v_source from the equations above
    v = strength / (2 * np.pi) * (Y - ys) / ((X - xs)**2 + (Y - ys)**2)

    return u, v
def infinite_vortex_vel(Gamma, a, X, Y):
    """
    Parameters
    ----------
    a: float
        Spacing between the vortices
    X: 2D Numpy array of floats
        x-coordinate of the mesh points.
    Y: 2D Numpy array of floats
        y-coordinate of the mesh points.

    Returns
    -------
    u: 2D Numpy array of floats
        x-component of the velocity vector field.
    v: 2D Numpy array of floats
        y-component of the velocity vector field.
    """
    # Here, input the equation for u_vortex from the equations above
    u = (Gamma/(2*a)) * (np.sinh(2*np.pi*Y/a)) / (np.cosh(2*np.pi*Y/a) - np.

    # Here, input the equation for v_vortex from the equations above
    v = -1* (Gamma/(2*a)) * (np.sin(2*np.pi*X/a)) / (np.cosh(2*np.pi*Y/a) -

    return u, v

vortStrength = 5
sstrength = -5

uVort, vVort = get_vortex_velocity(vortStrength,0,0,X,Y)
uSink, vSink = get_velocity(sstrength,0,0,X,Y)
uInfVort, vInfVort = infinite_vortex_vel(vortStrength,1,X,Y)


plt.figure(figsize=(20,20))

plt.subplot(2,2,1)
plt.title("Single Vortex")
plt.xlabel("X")
plt.ylabel("Y")
plt.streamplot(X, Y, uVort, vVort, density=2, linewidth=1, arrowsize=1, arro
plt.axis("equal")

plt.subplot(2,2,2)
plt.title("Single Vortex + Sink")
plt.xlabel("X")
plt.ylabel("Y")
plt.streamplot(X, Y, uVort + uSink, vVort + vSink, density=2, linewidth=1, a
```

```
plt.axis("equal")

sum_uVort = 0
sum_vVort = 0

nVortices = 12 #Number of vortices we want to plot
for xLoc in np.linspace(-5,5,nVortices):
    u, v = get_vortex_velocity(vortStrength,xLoc,0,X,Y)
    sum_uVort += u
    sum_vVort += v

plt.subplot(2,2,3)
plt.title("12 Vortices")
plt.xlabel("X")
plt.ylabel("Y")
plt.streamplot(X, Y, sum_uVort, sum_vVort, density=2, linewidth=1, arrowsize
plt.axis("equal");

plt.subplot(2,2,4)
plt.title("Inifinite Vortex Sheet")
plt.xlabel("X")
plt.ylabel("Y")
plt.streamplot(X, Y, uInfVort, vInfVort, density=2, linewidth=1, arrowsize=1
plt.axis("equal");
```



The first graph is of a vortex only, and is the basis for the rest of the plots. The second plot dipicts a source and a sink superimposed. The last two polots are of a finite sheet of 12 vorticies and of an infinite sheet approximation. The difference is that the finite sheet plot has edges, and from farther away the flow will look more and more like a single vortex. From close up, however, the flow appears to move to the right when above the x-axis, and to the left when below the x-axis. The infinite sheet approximation is a mathematical representation of an infinite sheet and displays the same left-right behavior as the finite sheet, but this time everywhere. As an approximation, it is still not perfect when close to the x-axis. By adjusting the spacing (closer spacing) and quantity of vorticies (more vortices total), we could make the finite sheet more closely approximate an infinite vortex sheet.

Vortex Panel Method

Problem 2 is focused on creating a vortex panel method to improve upon our previous method: the source panel method. Now, by imposing the Kutta condition, we can create an airfoil panel approximation that is capable of estimating lift.

In [13]:
```
#define each of the functions
class Panel:
    """
    Here, we are creating a panel object and all its necessary information.
    """

    def __init__(self, xa, ya, xb, yb):
        """
        Initialization of the panel.

        Here, we write a specific piece of code to be run everytime we creat
        will be run every time that we create a new panel.

        Our code needs to calculate the center point of the panel, the lengt

        Our code also needs to make space for the source strength, tangentia
        (which we will define for a specific panel later)

        Parameters:
        -----------
        xa: float
            x - coordinate of the first end point
        ya: float
            y - coordinate of the first end point
        xb: float
            x - coordinate of the second end point
        yb: float
            y - coordinate of the second end point
        xc: float
            x - coordinate of the center point of the panel
        yc: float
            y - coordinate of the center point of the panel
        length: float
            length of the panel
        beta: float
            orientation/angle of the panel

        These parameters are not defined until later. We set them equal to z
        sigma: float
            source sheet strength
        vt: float
            velocity tangential to the panel
        cp: float
            pressure coefficient


        """

        self.xa, self.ya = xa, ya           # Defines the first end point
        self.xb, self.yb = xb, yb           # Defines the second end point
```

```
        # Defining center point and panel parameters
        # You will need to define these yourself:
        self.xc, self.yc = (xa + xb) / 2, (ya + yb) / 2         # Control po
        self.length = math.sqrt((xb - xa)**2 + (yb - ya)**2)    # Length of

        # For the orientation of the panel (angle between x axis and the uni
        if xb - xa <= 0:
            self.beta = math.acos((yb - ya) / self.length)
        elif xb - xa > 0:
            self.beta = math.pi + math.acos(-(yb - ya) / self.length)

        # Location of the panel (we will use this later when we expand our a
        if self.beta <= math.pi:
            self.loc = 'upper'
        else:
            self.loc = 'lower'

        # Will need a value for theta
        self.theta = math.atan2(self.yc, self.xc)
        # We also need 3 more parameters, sigma, vt for tangential velocity,
        # Create these and set the equal to zero for now


        self.sigma = 0.0
        self.vt = 0.0
        self.cp = 0.0
def define_panels(x, y, N=40):
    """
    Discretizes the geometry into panels using the 'circle mapping' method.

    Parameters
    ----------
    x: 1D array of floats
        x-coordinate of the points defining the geometry.
    y: 1D array of floats
        y-coordinate of the points defining the geometry.
    N: integer, optional
        Number of panels;
        default: 40.

    Returns
    -------
    panels: 1D Numpy array of Panel objects
        The discretization of the geometry into panels.
    """
    R = (x.max() - x.min()) / 2                 # Radius of the circle, base
    x_center = (x.max() + x.min()) / 2          # X coordinate of center of

    x_circle = x_center + R * np.cos(np.linspace(0.0, 2 * math.pi, N + 1))
    # Here we define the x coordinates of the circle

    x_ends = np.copy(x_circle)                  # projection of the x-coord
    y_ends = np.empty_like(x_ends)              # initialization of the y-co

    x, y = np.append(x, x[0]), np.append(y, y[0])  # extend arrays using num
```

```
    # computes the y-coordinate of end-points
    I = 0
    for i in range(N):
        while I < len(x) - 1:
            if (x[I] <= x_ends[i] <= x[I + 1]) or (x[I + 1] <= x_ends[i] <=
                break
            else:
                I += 1
        a = (y[I + 1] - y[I]) / (x[I + 1] - x[I])
        b = y[I + 1] - a * x[I + 1]
        y_ends[i] = a * x_ends[i] + b
        #print(i)
        #print(I)
    x_ends[N] = x_ends[0]
    y_ends[N] = y_ends[0]

    panels = np.empty(N, dtype=object)
    for i in range(N):
        panels[i] = Panel(x_ends[i], y_ends[i], x_ends[i + 1], y_ends[i + 1]

    return panels
def vortex_integral_normal(p_i, p_j):
    """
    Evaluates the contribution of a source-panel at the center-point of anot
    in the normal direction.

    Parameters:
    -----------
    p_i: Panel object
        Panel on which the contribution is calculated.
    p_j: Panel object
        Panel from which the contribution is calculated.
    """

    def integrand(s):
        return (((p_i.xc - (p_j.xa - np.sin(p_j.beta)*s)) * (np.sin(p_i.beta
                 (p_i.yc - (p_j.ya + np.cos(p_j.beta)*s)) * (np.cos(p_i.beta
                ((p_i.xc - (p_j.xa - np.sin(p_j.beta)*s))**2 +
                 (p_i.yc - (p_j.ya + np.cos(p_j.beta)*s))**2))
    return integrate.quad(integrand, 0.0, p_j.length)[0]
def source_integral_normal(p_i, p_j):
    """
    Evaluates the contribution of a source-panel at the center-point of anot
    in the normal direction.

    Parameters:
    -----------
    p_i: Panel object
        Panel on which the contribution is calculated.
    p_j: Panel object
        Panel from which the contribution is calculated.
    """

    ## Fill in the equation below for the function integrand
    def integrand(s):
```

```
        return (((p_i.xc - (p_j.xa - math.sin(p_j.beta) * s)) * math.cos(p_i
                 (p_i.yc - (p_j.ya + math.cos(p_j.beta) * s)) * math.sin(p_i
                ((p_i.xc - (p_j.xa - math.sin(p_j.beta) * s))**2 +
                 (p_i.yc - (p_j.ya + math.cos(p_j.beta) * s))**2))
    return integrate.quad(integrand, 0.0, p_j.length)[0]
def analyze_panels(panels,uinf,alpha):
    """
    Here, we write some code to analyze our panels after they have been crea

    Creates a source influence matrix [A]


    Input: an array of panels created using the Panel function (panels) and

    """

    Num = len(panels)

    A_s = np.empty((Num, Num), dtype = float)
    A_v = np.empty((Num, Num), dtype = float)

    np.fill_diagonal(A_s, 0.5)
        # Whenever we have i = j, we have sigma(i)/2 or sigma(i)*0.5. Thus,
        # The diagonal of a matrix means i = j i.e (1,1), (2,2), etc etc.
        # Remember that this represents how much a source panel conributes t

    np.fill_diagonal(A_v, 0.0)
        # Here, we have the same thing, but for vortex contributions on norm

    # Create the source influence matrix [A_s] of the linear system
    # This represents how much a source panel contributes to OTHER panels no
    for i, p_i in enumerate(panels):
        for j, p_j in enumerate(panels):
            if i != j:
                A_s[i,j] = (0.5/np.pi) * source_integral_normal(p_i, p_j)
                # Here, we create a matrix called A_s to hold the source cor
                # Make sure that you change this to use our new functions ra

    # Create the vortex influence matrix [A_v]
    for i, p_i in enumerate(panels):
        for j, p_j in enumerate(panels):
            if i != j:
                A_v[i,j] = (-1 * 0.5/np.pi) * vortex_integral_normal(p_i,p_j
                # Here, we create a matrix called A_s to hold the source cor
                # Make sure that you change this to use our new functions ra

    A_s_norm = A_s
    A_v_norm = A_v

    # Kutta Condition:
    # First, lets create an array to hold all of our values.
    # This array should be of length N + 1 (the number of panels + 1)
    Kutta = np.empty(A_s.shape[0] + 1, dtype=float)

    # Next, we would like all the elements of our Kutta array (except the la
    # to be equal to the first and last values of our vortex contribution ma
```

```
    Kutta[:-1] = A_v[0, :] + A_v[-1, :]

    # Finally, we make the last element of our Kutta array equal to the sum
    # all the last elements of our source contribution matrix.
    Kutta[-1] = - np.sum(A_s[0, :] + A_s[-1, :])

    A = np.empty((Num+1, Num+1), dtype = float)

    # Enter the source contribution matrix
    # This takes up all but the last column and all but the last row
    # The vortex strength (gamma) and the kutta condition will take this pla
    A[:-1, :-1] = A_s_norm

    # Enter the vortex contribution array
    # Fills in the last column
    A[:-1, -1] = np.sum(A_v_norm, axis = 1)

    # Enter the Kutta array
    # Fills in the last row
    A[-1, :] = Kutta

    # Freestream Velocity and Matrix b
    # Lets start by creating an empty array b
    alpha = np.radians(alpha)
    b = np.empty(Num + 1, dtype = float)

    for i, panel in enumerate(panels):
        b[i] = - uinf * np.cos(alpha - panel.beta)

    # Freestream contribution on the Kutta condition
    b[-1] = - uinf * (math.sin(alpha - panels[0].beta) + math.sin(alpha - pa


    # Here we solve for the Strength sigma
    Strengths = np.linalg.solve(A,b)

    for i, panel in enumerate(panels):
        panel.sigma = Strengths[i]

    # The very last value in our strength array is the value gamma, or the c

    gamma = Strengths[-1]

    # Now tht we know what our source panel strengths are, as well as our vo
    # In order to computer the tangential velocity at each panel. We use a r

    # Computing Tangential Velocity
    A_t = np.empty((panels.size, panels.size+1), dtype = float)

    A_t[:, :-1] = A_v
    A_t[:, -1] = -np.sum(A_s, axis = 1)

    b_t = uinf * np.sin([alpha - panel.beta for panel in panels])

    vortex_strengths = np.append([panel.sigma for panel in panels], gamma)
```

```
    tan_vel = np.dot(A_t, vortex_strengths) + b_t

    for i, panel in enumerate(panels):
        panel.vt = tan_vel[i]


    # Finally, we need to compute the pressure coefficient.
    for panel in panels:
        panel.cp = 1 - (panel.vt/uinf)**2

    accuracy = sum([panel.sigma * panel.length for panel in panels])
    print('sum of singularity strengths: {:0.6f}'.format(accuracy))

    return panels

uinf = 2.2 #m/s
rho = 1.2 #kg/m^3
chord = 1.2 #m
mu = 1.591736e-5 #N*s/m^2 --> This part was interpolated to metric since we

alphaDeg = 8
alpha = alphaDeg*np.pi/180

Re = uinf * chord * rho/ mu
print("Reynolds number to be used in analysis: {:.3g}".format(Re))
```

```
Reynolds number to be used in analysis: 1.99e+05
```

These functions (and the class Panel) are very similar to the last project, and the Panel class is exactly the same. The only difference is that now, the Kutta condition is imposed and a nonzero circulation for panels is allowed. This affects the normal integral functions, which are changed accordingly. They Reynolds number is also calculated to be 1.99×10^5.

The next section analyzes the panels and creates the flow fields for each of the airfoils and the different angles of attack (0 and 8 degrees) and then creates the streamplots for each scenario.

In [14]:

```
#Analyze the panels and create flow fields

#define a new, smaller meshgrid
nx, ny = 80, 80  # number of points in the x and y directions
x_start, x_end = -0.5, 1.75
y_start, y_end = -0.3, 0.3
X, Y = np.meshgrid(np.linspace(x_start, x_end, nx),
                      np.linspace(y_start, y_end, ny))

folder = "airfoil-data"

filename = os.path.join(folder,"naca0018.txt")
x0018, z0018 = np.loadtxt(filename,dtype=float,skiprows=1,unpack=True)

filename = os.path.join(folder,"naca23012.txt")
x23012, z23012 = np.loadtxt(filename,dtype=float,skiprows=1,unpack=True)
```

```
#chord length isn't one, so mulitply everything by the chord lengthx
x0018 *= chord
z0018 *= chord
x23012 *= chord
z23012 *= chord

nPanels = 40
naca0018panelsa0 = define_panels(x0018,z0018,nPanels)
naca23012panelsa0 = define_panels(x23012,z23012,nPanels)

naca0018panelsaa = define_panels(x0018,z0018,nPanels)
naca23012panelsaa = define_panels(x23012,z23012,nPanels)

analyze_panels(naca0018panelsa0,uinf,0)
analyze_panels(naca23012panelsa0,uinf,0)

analyze_panels(naca0018panelsaa,uinf,alphaDeg)
analyze_panels(naca23012panelsaa,uinf,alphaDeg)

#function from aeropython to get the u and v velocity components in the mesh
def integral(x, y, panel, dxdz, dydz):
    """
    Evaluates the contribution of a panel at one point.

    Parameters
    ----------
    x: float
        x-coordinate of the target point.
    y: float
        y-coordinate of the target point.
    panel: Panel object
        Source panel which contribution is evaluated.
    dxdz: float
        Derivative of x in the z-direction.
    dydz: float
        Derivative of y in the z-direction.

    Returns
    -------
    Integral over the panel of the influence at the given target point.
    """
    def integrand(s):
        return (((x - (panel.xa - math.sin(panel.beta) * s)) * dxdz +
                 (y - (panel.ya + math.cos(panel.beta) * s)) * dydz) /
                ((x - (panel.xa - math.sin(panel.beta) * s))**2 +
                 (y - (panel.ya + math.cos(panel.beta) * s))**2) )
    return integrate.quad(integrand, 0.0, panel.length)[0]
def get_velocity_field(panels, uinf, alpha, X, Y):
    """
    Computes the velocity field on a given 2D mesh.

    Parameters
    ---------
    panels: 1D array of Panel objects
        The source panels.
```

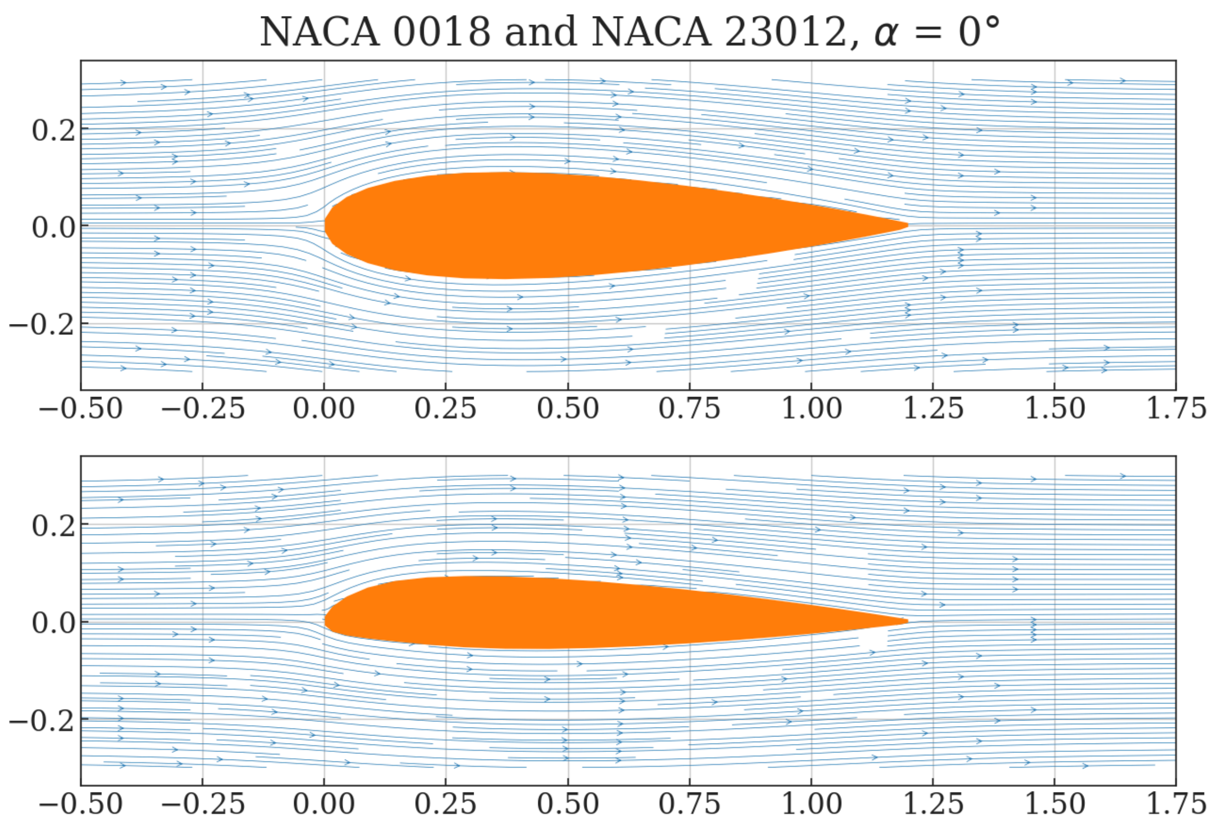
```
    freestream: Freestream object
        The freestream conditions.
    X: 2D Numpy array of floats
        x-coordinates of the mesh points.
    Y: 2D Numpy array of floats
        y-coordinate of the mesh points.

    Returns
    -------
    u: 2D Numpy array of floats
        x-component of the velocity vector field.
    v: 2D Numpy array of floats
        y-component of the velocity vector field.
    """
    # freestream contribution
    u = uinf * math.cos(alpha) * np.ones_like(X, dtype=float)
    v = uinf * math.sin(alpha) * np.ones_like(X, dtype=float)
    # add the contribution from each source (superposition powers!!!)
    vec_intregral = np.vectorize(integral)
    for panel in panels:
        u += panel.sigma / (2.0 * math.pi) * vec_intregral(X, Y, panel, 1.0,
        v += panel.sigma / (2.0 * math.pi) * vec_intregral(X, Y, panel, 0.0,

    return u, v


u0018aa, v0018aa = get_velocity_field(naca0018panelsaa,uinf,alpha,X,Y)
u23012aa, v23012aa = get_velocity_field(naca23012panelsaa,uinf,alpha,X,Y)

u0018a0, v0018a0 = get_velocity_field(naca0018panelsa0,uinf,0,X,Y)
u23012a0, v23012a0 = get_velocity_field(naca23012panelsa0,uinf,0,X,Y)
```

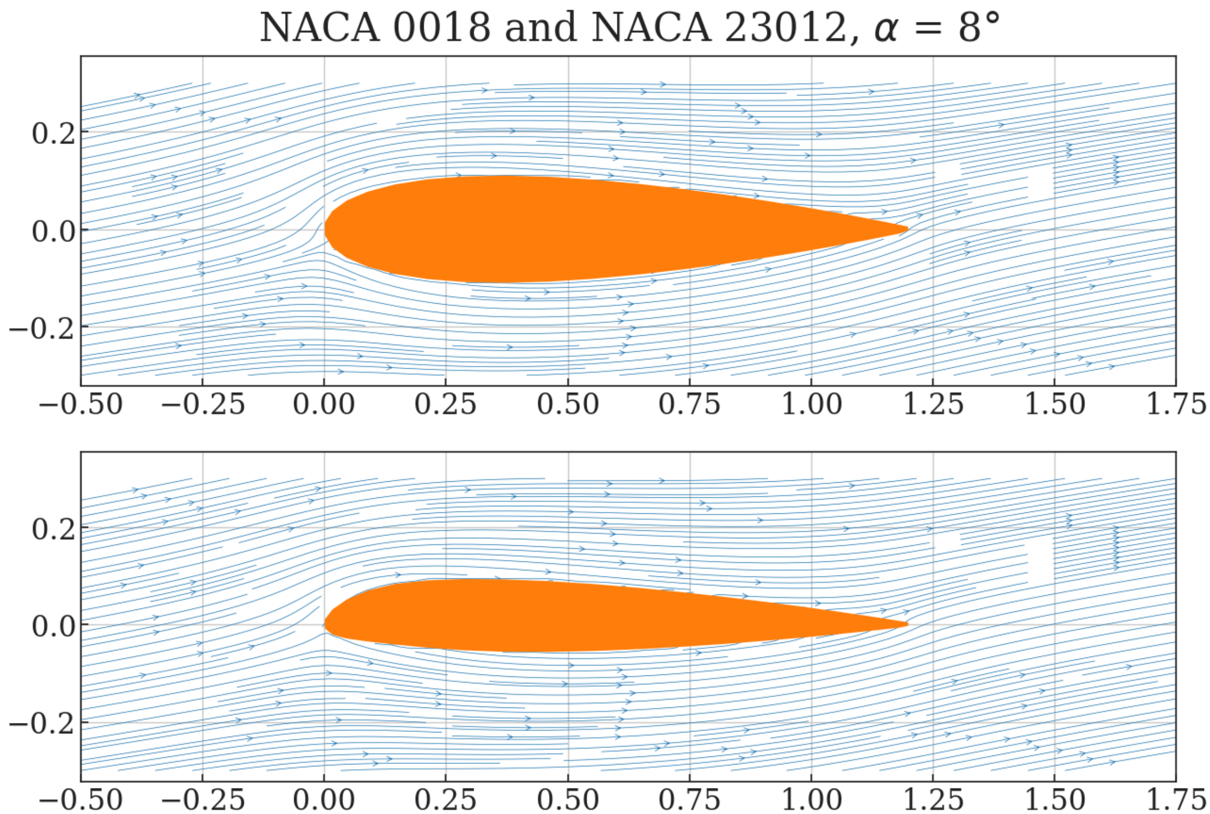
```
sum of singularity strengths: 0.017645
sum of singularity strengths: 0.013009
sum of singularity strengths: 0.018362
sum of singularity strengths: 0.005262
```

In [15]:

```
#plot everything

plt.figure(figsize=(15,10))
plt.subplot(2,1,1)
plt.title(r"NACA 0018 and NACA 23012, $\alpha$ = 0$\degree$")
plt.streamplot(X,Y,u0018a0,v0018a0,density=2,linewidth=0.5,arrowsize=1,arrow
plt.fill([panel.xc for panel in naca0018panelsa0],
            [panel.yc for panel in naca0018panelsa0],
            color=colors[1], linestyle='solid', linewidth=2, zorder=2)
plt.axis("equal");

plt.subplot(2,1,2)
plt.streamplot(X,Y,u23012a0,v23012a0,density=2,linewidth=0.5,arrowsize=1,arr
plt.fill([panel.xc for panel in naca23012panelsa0],
            [panel.yc for panel in naca23012panelsa0],
            color=colors[1], linestyle='solid', linewidth=2, zorder=2)
plt.axis("equal");

plt.figure(figsize=(15,10))
```

```
plt.subplot(2,1,1)
plt.title(r"NACA 0018 and NACA 23012, $\alpha$ = {}$\degree$".format(alphaDe
#plt.plot(x0018,z0018,color = colors[1])
plt.streamplot(X,Y,u0018aa,v0018aa,density=2,linewidth=0.5,arrowsize=1,arrow
plt.fill([panel.xc for panel in naca0018panelsaa],
            [panel.yc for panel in naca0018panelsaa],
            color=colors[1], linestyle='solid', linewidth=2, zorder=2)
plt.axis("equal")

plt.subplot(2,1,2)
plt.streamplot(X,Y,u23012aa,v23012aa,density=2,linewidth=0.5,arrowsize=1,arr
plt.fill([panel.xc for panel in naca23012panelsaa],
            [panel.yc for panel in naca23012panelsaa],
            color=colors[1], linestyle='solid', linewidth=2, zorder=2)
plt.axis("equal");
```





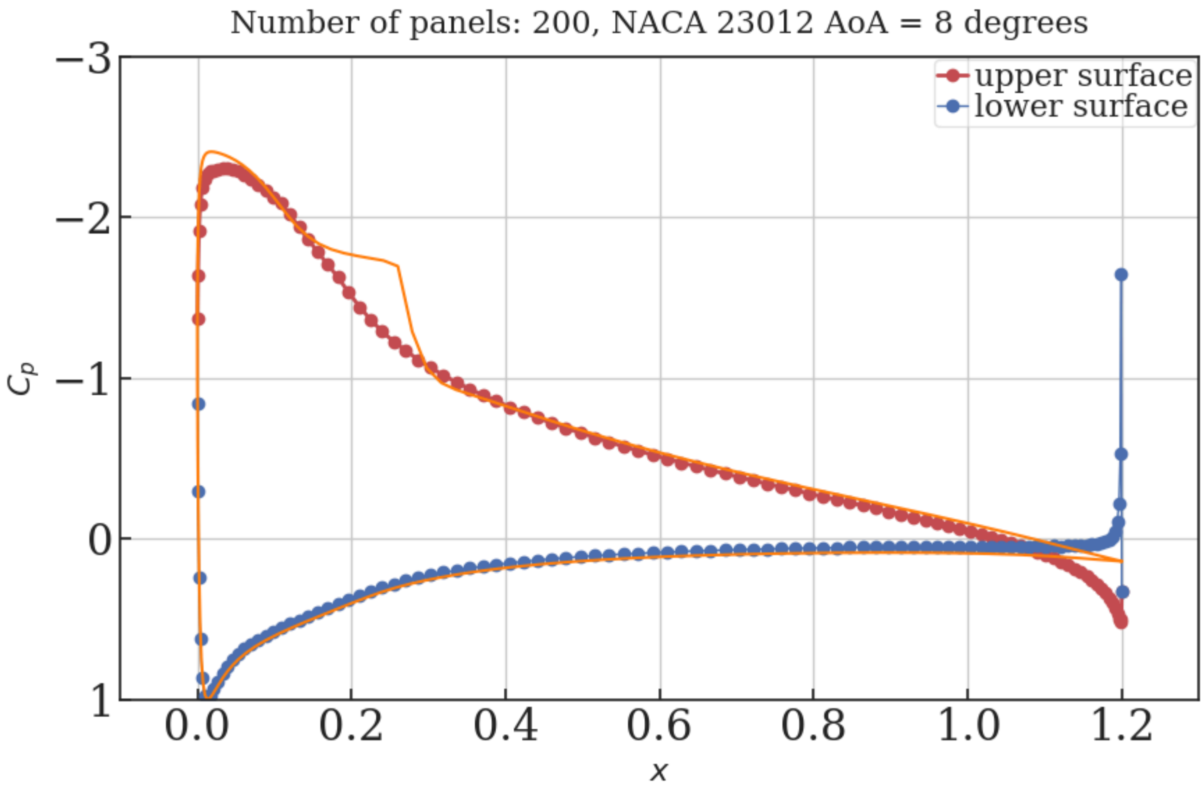
These plots show how the NACA 0018 airfoil and NACA 23012 airfoil behave at different angles of attack. Also, since NACA 23012 is a cambered airfoil, it has a noticeably asymmetric flow around it when compared to the symmetric NACA 0018 airfoil.

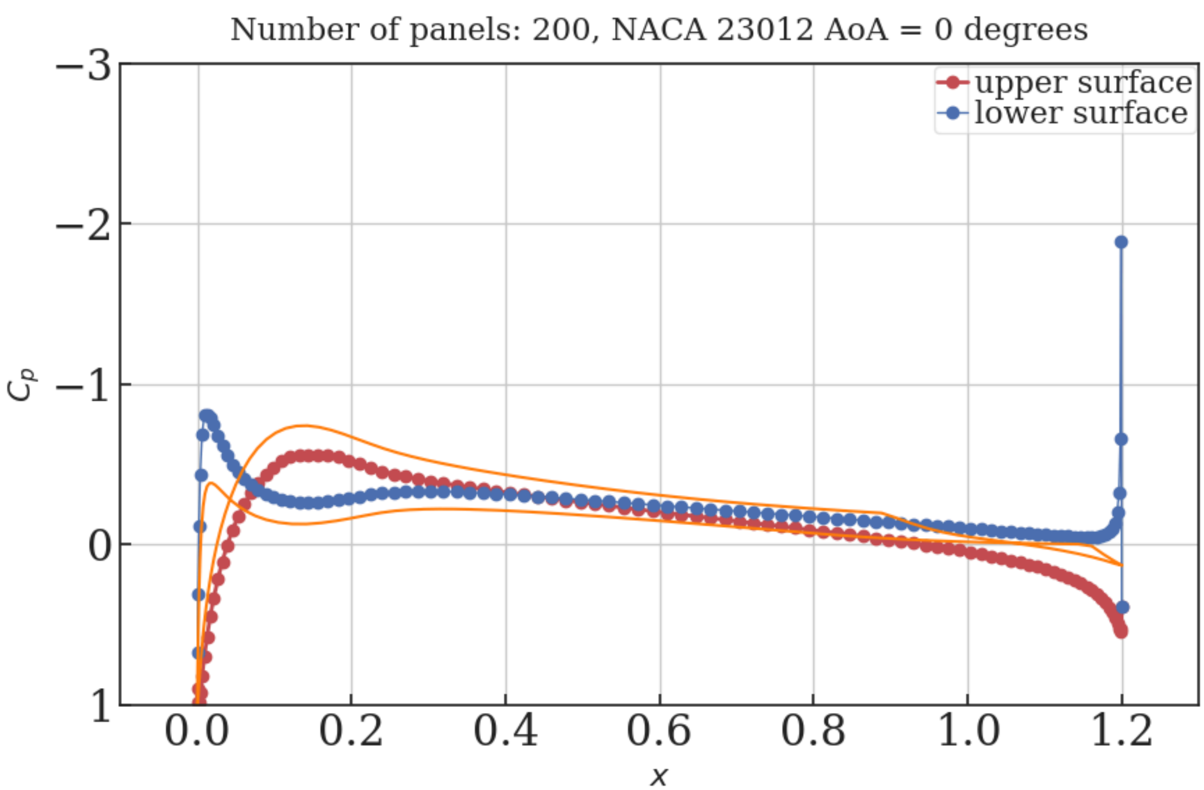
I also compared the pressure distribution graphs, which are plotted next.

In [20]:
```
#plot pressure graphs
nPanels = 200
naca23012panelsaa = define_panels(x23012,z23012,nPanels)
analyze_panels(naca23012panelsaa,uinf,alphaDeg)
naca23012panelsa0 = define_panels(x23012,z23012,nPanels)
analyze_panels(naca23012panelsa0,uinf,0)

filename = os.path.join(folder,"naca23012a8V.txt")
xfoil23012xCpVa8, xfoil23012CpVa8 = np.loadtxt(filename,dtype=float,skiprows
xfoil23012xCpVa8 *= chord

filename = os.path.join(folder,"naca23012a0V.txt")
xfoil23012xCpVa0, xfoil23012CpVa0 = np.loadtxt(filename,dtype=float,skiprows
xfoil23012xCpVa0 *= chord
#filename = os.path.join(folder,"naca23012a8.txt")
#xfoil23012xCpI, xfoil23012CpI = np.loadtxt(filename,dtype=float,skiprows=1,

#plot for alpha = 8
plt.figure(figsize=(10, 6))
plt.grid()
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$C_p$', fontsize=16)
plt.plot([panel.xc for panel in naca23012panelsaa if panel.loc == 'upper'],
            [panel.cp for panel in naca23012panelsaa if panel.loc == 'upper'
```

```
            label='upper surface',
            color='r', linestyle='-', linewidth=2, marker='o', markersize=6)
plt.plot([panel.xc for panel in naca23012panelsaa if panel.loc == 'lower'],
            [panel.cp for panel in naca23012panelsaa if panel.loc == 'lower'
            label= 'lower surface',
            color='b', linestyle='-', linewidth=1, marker='o', markersize=6)
plt.legend(loc='best', prop={'size':16})
plt.xlim(-0.1, 1.3)
plt.ylim(1.0, -3.0)
plt.title('Number of panels: {}, NACA 23012 AoA = 8 degrees'.format(naca2301
plt.plot(xfoil23012xCpVa8,xfoil23012CpVa8,color = colors[1])
plt.grid(True)

#plot for alpha = 0
plt.figure(figsize=(10, 6))
plt.grid()
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$C_p$', fontsize=16)
plt.plot([panel.xc for panel in naca23012panelsa0 if panel.loc == 'upper'],
            [panel.cp for panel in naca23012panelsa0 if panel.loc == 'upper'
            label='upper surface',
            color='r', linestyle='-', linewidth=2, marker='o', markersize=6)
plt.plot([panel.xc for panel in naca23012panelsa0 if panel.loc == 'lower'],
            [panel.cp for panel in naca23012panelsa0 if panel.loc == 'lower'
            label= 'lower surface',
            color='b', linestyle='-', linewidth=1, marker='o', markersize=6)
plt.legend(loc='best', prop={'size':16})
plt.xlim(-0.1, 1.3)
plt.ylim(1.0, -3.0)
plt.title('Number of panels: {}, NACA 23012 AoA = 0 degrees'.format(naca2301
plt.plot(xfoil23012xCpVa0,xfoil23012CpVa0,color = colors[1])
plt.grid(True);
```

```
sum of singularity strengths: 0.001073
sum of singularity strengths: 0.002827
```





In [17]:
```
#Calculate coefficient of lift
def getClCd():

    upperCp = [p.cp for p in naca23012panelsa0 if p.loc == "upper"]
    lowerCp = [p.cp for p in naca23012panelsa0 if p.loc == "lower"]

    upperX = [p.xa for p in naca23012panelsa0 if p.loc == "upper"]
```

```
    lowerX = [p.xa for p in naca23012panelsa0 if p.loc == "lower"]

    z23012up = []
    x23012up = []
    z23012low = []
    x23012low = []
    for i,element in enumerate(z23012):
        if element > 0:
            z23012up = np.append(z23012up,element)
            x23012up = np.append(x23012up,x23012[i])
        else:
            z23012low = np.append(z23012low,element)
            x23012low = np.append(x23012low,x23012[i])

    upperZremapped = np.interp(upperX,x23012up,z23012up)
    lowerZremapped = np.interp(lowerX,x23012low,z23012low)

    CnUpperInt = np.trapz(upperCp,upperX)
    CnLowerInt = np.trapz(lowerCp,lowerX)

    Cn = CnLowerInt - CnUpperInt

    CaUpperInt = np.trapz(np.multiply(upperCp,np.gradient(upperZremapped,upp
    CaLowerInt = np.trapz(np.multiply(lowerCp,np.gradient(lowerZremapped,low

    Ca = CaUpperInt - CaLowerInt

    Cl = Cn*np.cos(alpha) - Ca*np.sin(alpha)
    Cd = Cn*np.sin(alpha) + Ca*np.cos(alpha)

    return Cl, Cd
#DOESN'T WORK, NOT NECESSARY FOR PROJECT, DISREGARD
```

These graphs show the Cp distribution across the NACA 23012 airfoil as calculated using the vortex panel method compared to XFOIL, with the XFOIL plot in orange. For the 8 degree angle of attack plot, our method matches up quite will with XFOIL. The 0 degree angle of attack plots also looks similar to each other, but the magnitudes are a bit off. Additionally, the vortex panel method seems to have a bit of a spike in CP at the end.

A vortex panel method is capable of producing lift because of the Kutta lift condition, which has a circulation term contained within it. It is not possible to produce lift unless there is some source of circulation; for us, a vortex. In real life, the flow around an airfoil can be mathematically modeled as a combination of sources, sinks, and vortices, where the vortices are responsible for lift. Computationally, the source vorticies factor in to the tangential and normal contributions for each panel, which in turn will affect the overall fluid stream.

In [18]:
```
#Use xfoil to find Cl and Cd for a variety of conditions and airfoils
Cl2412Re3 = 0.8069
Cd2412Re3 = 0.00677

Cl4412Re3 = 1.0345
```

```
Cd4412Re3 = 0.00632

Cl23012Re3 = 0.6794
Cd23012Re3 = 0.00645

Cl2412Re9 = 0.7996
Cd2412Re9 =  0.00628

Cl4412Re9 = 1.0368
Cd4412Re9 = 0.00607

Cl23012Re9 = 0.6994
Cd23012Re9 = 0.00539

#Lift and Drag calculations:
def getLift(Cl,rho,c,v):
    return (Cl/2) * c * rho * v**2
def getDrag(Cd,rho,c,v):
    return (Cd/2) * c * rho * v**2

rho = 2.0481e-3 #slugs/ft^3
mu = 3.636e-7 #lb * s/ft^2
c = 7 #ft

re1 = 3e6
re2 = 9e6

v1 = re1/(rho*c/mu)
v2 = re2/(rho*c/mu)

L2412Re3 = getLift(Cl2412Re3,rho,c,v1)
D2412Re3 = getDrag(Cd2412Re3,rho,c,v1)

L4412Re3 = getLift(Cl4412Re3,rho,c,v1)
D4412Re3 = getDrag(Cd4412Re3,rho,c,v1)

L23012Re3 = getLift(Cl23012Re3,rho,c,v1)
D23012Re3 = getDrag(Cd23012Re3,rho,c,v1)

L2412Re9 = getLift(Cl2412Re9,rho,c,v2)
D2412Re9 = getDrag(Cd2412Re9,rho,c,v2)

L4412Re9 = getLift(Cl4412Re9,rho,c,v2)
D4412Re9 = getDrag(Cd4412Re9,rho,c,v2)

L23012Re9 = getLift(Cl23012Re9,rho,c,v2)
D23012Re9 = getDrag(Cd23012Re9,rho,c,v2)

def printLD(liftforce,dragforce,airfoil):
    print("---------------------------------")
    print("Lift of {}: {:.4g} lbf".format(airfoil,liftforce))
    print("Drag of {}: {:.4g} lbf".format(airfoil,dragforce))
    print("---------------------------------")

printLD(L2412Re3,D2412Re3,"NACA 2412, 3e6")
printLD(L4412Re3,D4412Re3,"NACA 4412, 3e6")
```

```
printLD(L23012Re3,D23012Re3,"NACA 23012, 3e6")
printLD(L2412Re9,D2412Re9,"NACA 2412, 9e6")
printLD(L4412Re9,D4412Re9,"NACA 4412, 9e6")
printLD(L23012Re9,D23012Re9,"NACA 23012, 9e6")
```

```
--------------------------------
Lift of NACA 2412, 3e6: 33.48 lbf
Drag of NACA 2412, 3e6: 0.2809 lbf
--------------------------------
--------------------------------
Lift of NACA 4412, 3e6: 42.93 lbf
Drag of NACA 4412, 3e6: 0.2623 lbf
--------------------------------
--------------------------------
Lift of NACA 23012, 3e6: 28.19 lbf
Drag of NACA 23012, 3e6: 0.2677 lbf
--------------------------------
--------------------------------
Lift of NACA 2412, 9e6: 298.6 lbf
Drag of NACA 2412, 9e6: 2.345 lbf
--------------------------------
--------------------------------
Lift of NACA 4412, 9e6: 387.2 lbf
Drag of NACA 4412, 9e6: 2.267 lbf
--------------------------------
--------------------------------
Lift of NACA 23012, 9e6: 261.2 lbf
Drag of NACA 23012, 9e6: 2.013 lbf
--------------------------------
```

For finding the maximum L/D, I iterated over different angles of attack in xfoil by hand to see where the highest L/D was for each airfoil.

NACA 2412: MAx L/D ≈ 135.61 at an angle of attack of \approx 7.95\degree\newline
NACA 4412: Max L/D ≈ 173.42 at an angle of attack of \approx 4.75\degree\newline
NACA 23012: Max L/D ≈ 155.89 at an angle of attack of \approx 8.05\degree\newline