Source Panel Method

Nicholas Richmond

Student ID: 919578115

Project #: 3

Date: 11/8/24

EAE 127: Applied Aerodynamics

University of California, Davis

In [186…

```
#standard imports and setups
import pandas as pd #type: ignore
import numpy as np #type: ignore
import os.path
import matplotlib.pyplot as plt #type: ignore
import matplotlib.lines as mlines
import math
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': 15, #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: Source Panel Method

This problem is an introduction to how the panel method works, and involves setting up a few things in python. Namely, the class `Panel` and the functions `integral_normal()` and `integral_tangential()`.

In [187…

```
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 create a panel object. In a general sense
        will be run every time that we create a new panel.

        Our code needs to calculate the center point of the panel, the length, and its angle.

        Our code also needs to make space for the source strength, tangential velocity, and pressure coefficient
        (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 zero below.
        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 point or center point (How do you find
        self.length = math.sqrt((xb - xa)**2 + (yb - ya)**2)    # Length of the panel (How do you find the dista

        # For the orientation of the panel (angle between x axis and the unit vector normal to the panel)
        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 analys to airfoils)
        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)

        if self.theta < 0:
            self.theta += 2*np.pi
        # We also need 3 more parameters, sigma, vt for tangential velocity, and cp for pressure distribution.
        # Create these and set the equal to zero for now


        self.sigma = 0.0
        self.vt = 0.0
        self.cp = 0.0

def integral_normal(p_i, p_j):
    """
    Evaluates the contribution of a panel at the center-point of another,
    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 - math.sin(p_j.beta) * s)) * math.cos(p_i.beta) +
                 (p_i.yc - (p_j.ya + math.cos(p_j.beta) * s)) * math.sin(p_i.beta)) /
                ((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 integral_tangential(p_i, p_j):
    """
    Evaluates the contribution of a panel at the center-point of another,
    in the tangential direction.

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

    Returns
    -------
    Integral over the panel at the center point of the other.
    """

    # You will need to write in the tangetial velocity equation below.
    # Remember that we only need the terms within the integrand, and we can leave
    # the freestream velocity (U infinity) term for later.
    def integrand(s):
        return ((-(p_i.xc - (p_j.xa - math.sin(p_j.beta) * s)) * math.sin(p_i.beta) +
                 (p_i.yc - (p_j.ya + math.cos(p_j.beta) * s)) * math.cos(p_i.beta)) /
                ((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]
```

Panel Class:

The Panel class is an object created to help define different panels. Each new panel that we define in the future will have a few attributes that are gathered from the x and y coordinates of each end of the panel. Things like the angle between the normal of the panel and the x-axis, whether the panel is on the upper or lower surface of the airfoil and the length all are designated as properties of the panel, which can be called with syntax in the form of `Panel.attribute_of_interest`.

The Panel class takes in the coordinates of both ends of the desired panel. Using trigonometry, most other aspects and metrics about the panel can be derived and assigned. The remaining aspects (such as coefficient of pressure) can be found by picking a freestream velocity and then iterating over every panel to determine the effects of all panels on each other. It is for this reason that we need the tangentail and normal integral functions (described next).

The panel method is useful to us for aerodynamics analysis because it allows us to quickly create and model many panels, each with their own unique attirbutes. The ability to create any number of panels and to keep track of them in an organized and systematic way is why creating an object for panels is a good choice.

Normal Integral Fuction:

The purpose of the normal integral function is to determine the contribution of any source panel on the center point of any other source panel, in the normal direction. It takes in the two panels in question, and finds the contribution of first panel at the center point of the second panel, and returns that contribution. It uses the locations of the panel center points, the locations of the panel ends, and the beta angles of the panels using the `Panel` class.

Tangential Integral Function

The purpose of the tangential integral function is very similar to that of the normal integral function. The only difference is the direction of the contribution. In the normal integral function, the contribution is found in the normal direction; in the tangential integral function, the contribution is found in the tangential direction.

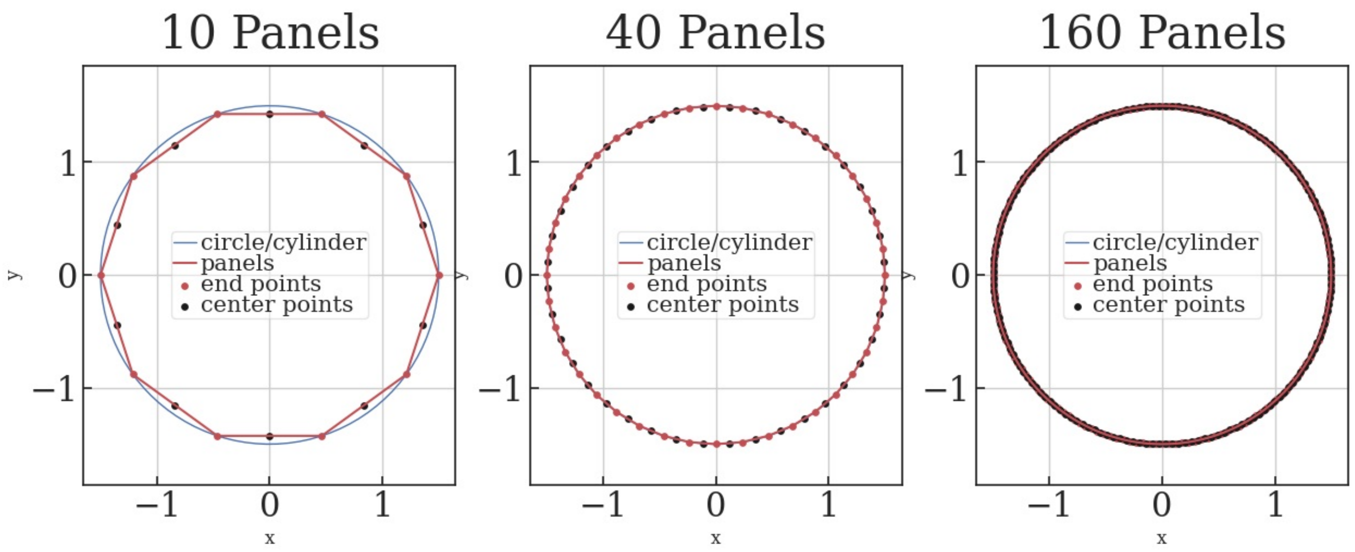
Problem 2: Flow Over a Non-Rotating Cylinder

This problem asks us to analyze the flow over a nonrotating cylinder using the panel discretization method using different numbers of panels.

In [188…

```
#create the geometry of the circle and the x-y coordinates of the panels
N_panels = 6            # We are going to use 6 panels in this quick test to see how our panel function/object


R = 1.5
x_circle = R * np.cos(np.linspace(0.0, 2 * math.pi, 100))
y_circle = R * np.sin(np.linspace(0.0, 2 * math.pi, 100))

#function to generate any number of panels and their coordinates
def discretizeCylinder(numPanels):

    x_points = R * np.cos(np.linspace(0.0, 2 * math.pi, numPanels + 1))
    y_points = R * np.sin(np.linspace(0.0, 2 * math.pi, numPanels + 1))
    # Here we just create some information that describes a circle, but we will also use these points
    # to describe our panel end points.

    cylinder_panels = np.empty(numPanels, dtype = object)
    # The line above creates an array called cylinder_panels, and designates that the type of data we are going
    # In other words we will use this array to store all of the panel objects we are about to create.

    # Here is where we will do something new, using our Panel code we created above:
    for i in range(numPanels):
        cylinder_panels[i] = Panel(x_points[i], y_points[i], x_points[i+1], y_points[i+1])

    return x_points, y_points, cylinder_panels

def plotPaneledCircle(cylinder_panels):

    plt.grid()
    plt.xlabel('x', fontsize = 12)
    plt.ylabel('y', fontsize = 12)

    plt.plot(x_circle, y_circle, label = 'circle/cylinder', color = 'b', linewidth = 1)
    # Plotting our circle/the geometry we would like to discretize

    plt.plot(x_points, y_points, label = 'panels', color = 'r')
    # Plotting our end points / panels
    # Here, we are plotting the exact same points that we provided to our Panel function.
    # Remember that we are using these as the end points of our panels

    plt.scatter([p.xa for p in cylinder_panels], [p.ya for p in cylinder_panels], label = 'end points', color =
    # This code may look a little bit odd to you. The code within the brackets accesses the information
    # inside of the variable 'cylinder_panels'. For each panel inside of 'cylinder_panels', we plot
    # the xa and ya variables associated with those panels.

    plt.scatter([p.xc for p in cylinder_panels], [p.yc for p in cylinder_panels], label = 'center points', colo
    # The same is done here as above, except we use the center points instead of the end points.
    # Using the code here we can access any of the information within a specific Panel object.
    plt.axis("equal")
    plt.legend(loc = 'best')


plt.figure(figsize=(15,5))
x_points, y_points, cylinder_panels = discretizeCylinder(10)
plt.subplot(1,3,1)
plt.grid()
plt.title("10 Panels")
plotPaneledCircle(cylinder_panels)

x_points, y_points, cylinder_panels = discretizeCylinder(40)
plt.subplot(1,3,2)
plt.grid()
```

```python
plt.title("40 Panels")
plotPaneledCircle(cylinder_panels)

plt.subplot(1,3,3)
plt.grid()
plt.title("160 Panels")
x_points, y_points, cylinder_panels = discretizeCylinder(160)
plotPaneledCircle(cylinder_panels)
plt.show()
```



These three plots show how more panels can create a better approximation of a circle using flat elements. Even at just 40 elements, the approximation to a cylinder is very good.

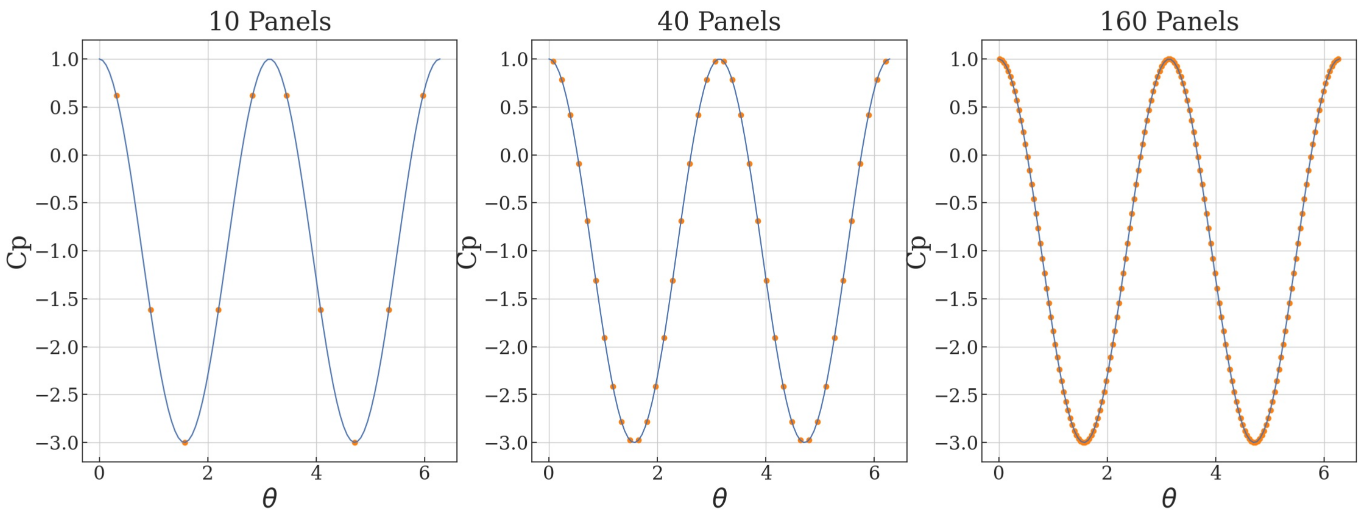
In the next section we will analyze each of these cases to determine the strength of each panel and confirm if each distribution creates a closed body.

In [189…

```python
#Analyze different discretizations of a cylinder using the panel method
#adds other important information about each panel to each panel class.
def analyze_panels(panels, u_inf):
    """"
    Here, we write some code to analyze our panels after they have been created.

    Creates a source influence matrix [A]


    Input: an array of panels created using the Panel function (panels) and a freestream velocity (u_inf).

    """

    Num = len(panels)

    # First we need the Normal Velocity Calculations

    A_n = np.empty((Num, Num), dtype = float)
    np.fill_diagonal(A_n, 0.5)
        # Whenever we have i = j, we have sigma(i)/2 or sigma(i)*0.5. Thus, on our diagonal for matrix A we shou
        # The diagonal of a matrix means i = j i.e (1,1), (2,2), etc etc.

    # Create the source influence matrix [A] of the linear system
    for i, p_i in enumerate(panels):
        for j, p_j in enumerate(panels):
            if i != j:
                A_n[i,j] = (0.5/math.pi) * integral_normal(p_i, p_j)

    # Create the right hand side [b] of the linear system
    b_n = - u_inf * np.cos([p.beta for p in panels])

    sigma = np.linalg.solve(A_n,b_n)

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

    # ====================================================================== #
    # Now we need the Tangential Velocity Calculations
    A_t = np.empty((Num, Num), dtype = float)
    np.fill_diagonal(A_t, 0.0)

    # Create the source influence matrix [A] of the linear system
    # STUDENTS WILL FILL THIS IN DELETE LATER
    for i, p_i in enumerate(panels):
```

```python
        for j, p_j in enumerate(panels):
            if i!=j:
                A_t[i,j] = (0.5 / math.pi) * integral_tangential(p_i, p_j)
    # Create the right hand side [b] of the linear system
    # STUDENTS ALSO FILL THIS IN
    b_t = -u_inf * np.sin([p.beta for p in panels])

    # Finally, we compute the tangential velocity:
    # STUDENTS ALSO FILL THIS IN
    vt = np.dot(A_t, sigma) + b_t

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

    # STUDENTS FILL THIS IN
    # Finally, lets use our tangential velocity to calculate surface pressure Cp:


    for panel in panels:
        panel.cp = 1.0 - (panel.vt/u_inf)**2

    print('Panel Analysis for {} Panels Complete!'.format(Num),end="\r")


u_inf = 16.4 #5 meters per second to feet per second
x_points10, y_points10, cylinder_panels10 = discretizeCylinder(10)
analyze_panels(cylinder_panels10, u_inf)
sumSource10 = 0
for panel in cylinder_panels10:
    sumSource10 += panel.sigma*panel.length

x_points40, y_points40, cylinder_panels40 = discretizeCylinder(40)
analyze_panels(cylinder_panels40, u_inf)
sumSource40 = 0
for panel in cylinder_panels40:
    sumSource40 += panel.sigma*panel.length

x_points160, y_points160, cylinder_panels160 = discretizeCylinder(160)
analyze_panels(cylinder_panels160, u_inf)
sumSource160 = 0

for panel in cylinder_panels160:
    sumSource160 += panel.sigma*panel.length
print("\nSum of sources and sinks:\n10 Panel Cylinder: {:.5g}\n40 Panel Cylinder: {:.5g}\n160 Panel Cylinder: {
#print(sumSource10, sumSource40, sumSource160)
cp_analytical = 1.0 - 4 * (y_circle / R)**2
length = (len(x_circle))
theta = np.zeros((length))

for i in range(0, length):
    theta[i] = math.atan2(y_circle[i], x_circle[i])
    if theta[i] < 0:
        theta[i] += 2*np.pi

plt.figure(figsize=(27,9))
plt.subplot(1,3,1)
plt.scatter([panel.theta for panel in cylinder_panels10],[panel.cp for panel in cylinder_panels10],label = "Coe
plt.plot(theta, cp_analytical,label='analytical',color='b', linestyle='-', linewidth=1.5, zorder=1);
plt.title("10 Panels")
plt.xlabel(r"$\theta$")
plt.ylabel("Cp")

plt.subplot(1,3,2)
plt.scatter([panel.theta for panel in cylinder_panels40],[panel.cp for panel in cylinder_panels40],label = "Coe
plt.plot(theta, cp_analytical,label='analytical',color='b', linestyle='-', linewidth=1.5, zorder=1);
plt.title("40 Panels")
plt.xlabel(r"$\theta$")
plt.ylabel("Cp")

plt.subplot(1,3,3)
plt.scatter([panel.theta for panel in cylinder_panels160],[panel.cp for panel in cylinder_panels160],label = "C
plt.plot(theta, cp_analytical,label='analytical',color='b', linestyle='-', linewidth=1.5, zorder=1);
plt.title("160 Panels")
plt.xlabel(r"$\theta$")
plt.ylabel("Cp");
```

```
Panel Analysis for 160 Panels Complete!
Sum of sources and sinks:
10 Panel Cylinder: 2.1316e-14
40 Panel Cylinder: 7.9936e-15
160 Panel Cylinder: -1.3323e-15
```



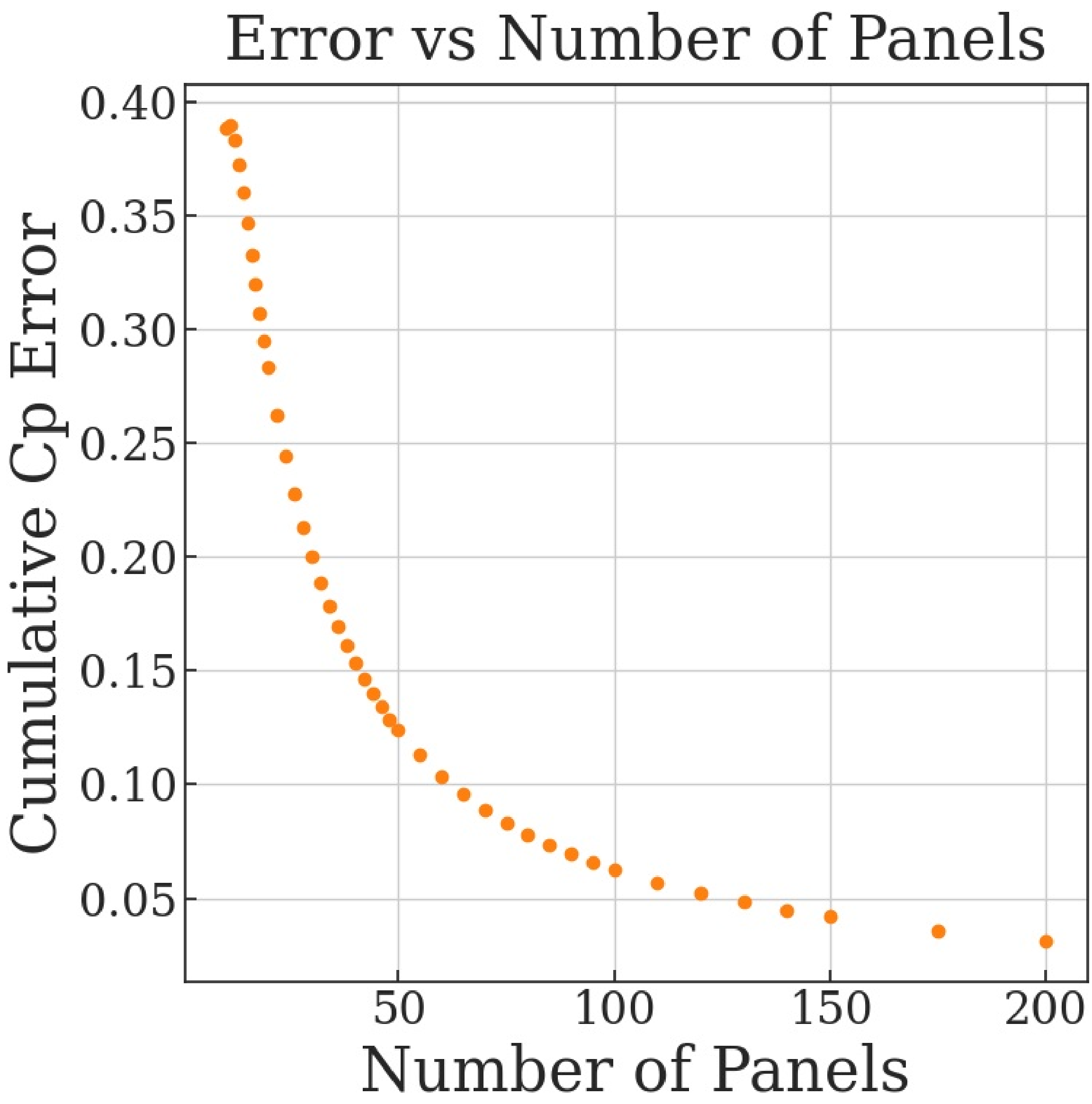
The strengths cancel out in accordance with Anderson 3.157. Interestingly, when summing the strengths of the panels, if an array of each strength was first created (using `np.append()`), there were increasingly large precision errors with panel counts above 127. While I did not try to test it too much, it seems like there is a large floating point innacuracy within `np.append` and I am not sure why.

The next step is to quantify the errors of each method by finding the difference in the integrated areas of both curves at various panel counts.

In [190…

```
#Find the error of the panel method's cp vs the analytical cylinder cp

def getError(numPanels):
    x_points, y_points, cylinder_panels = discretizeCylinder(numPanels)
    analyze_panels(cylinder_panels, u_inf)

    error = np.trapz(cp_analytical,theta) - np.trapz([p.cp for p in cylinder_panels],[p.theta for p in cylinder_
    return abs(error)

plt.figure()
i = 10
while i < 201:
    error = getError(i)
    plt.plot(i,error, marker = 'o',linestyle = '-',linewidth = 1,color = colors[1])
    if i < 20:
        i += 1
    elif i < 50:
        i += 2
    elif i < 100:
        i += 5
    elif i < 150:
        i += 10
    elif i < 200:
        i += 25
    elif i < 300:
        i += 50
    else:
        i += 100

plt.title("Error vs Number of Panels")
plt.xlabel("Number of Panels")
plt.ylabel("Cumulative Cp Error")
plt.show()
```

```
Panel Analysis for 200 Panels Complete!
```



This plot displays the absolute value of the error between the cumulative C_p of various panel approximations and the cumulative analytical C_p of the cylinder. There is a very clear trend of decreasing error as the number of panels increases, which appears to be exponential decay. This value approaches zero as the number of panels increases, but never actually reaches it. Also, for large panel numbers, the amount of time it took to analyze each new cylinder discretization increased drastically (probably exponentially). Thus, there is a certain discretization that is the most efficient in terms of time taken to run and acceptable error.

Problem 3: Symmetric and Cambered Airfoil Panel Methods

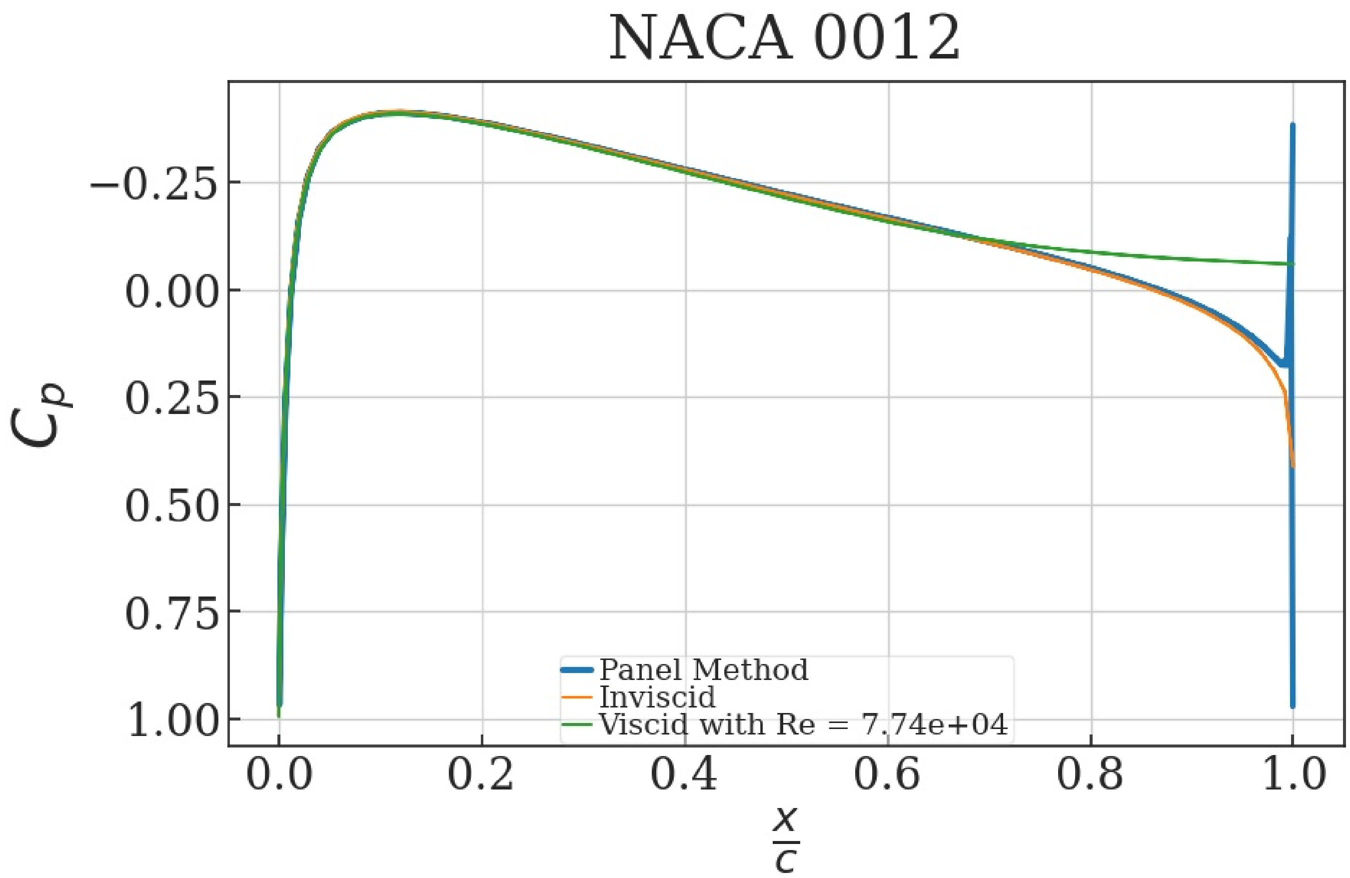
In [195…

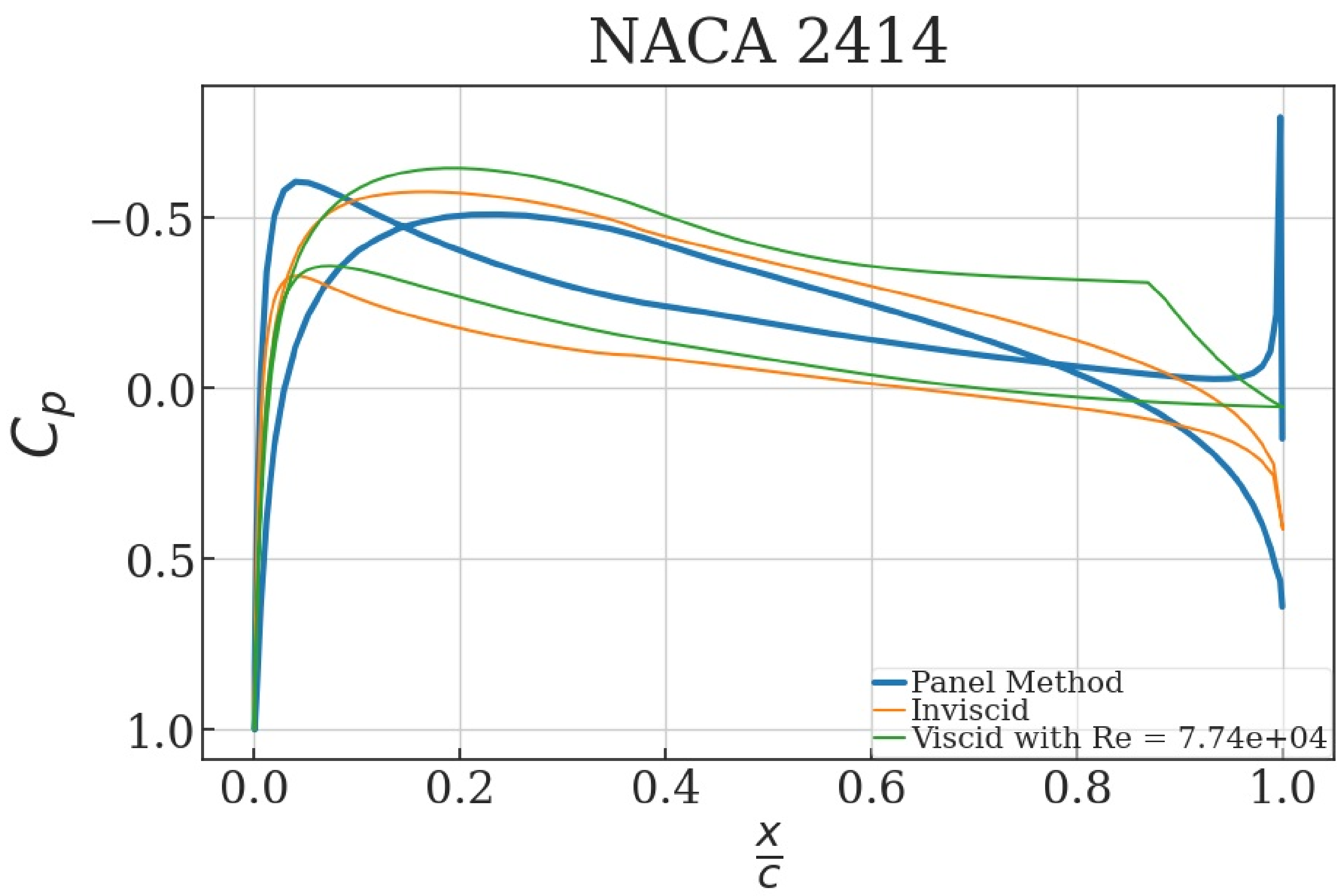
```
#airfoil panel method analysis

def naca4(m_in,p_in,t_in): #creates the x and z arrays of the airfoil and plots them given 4 digit naca number
    stepsize = 0.005
    m = m_in/100
    p = p_in/10
    t = t_in/100
    x = np.arange(0,1+stepsize,stepsize)
    upper = np.empty(0)
```

```
    lower = np.empty(0)
    camber = np.empty(0)

    if (m == 0) and (p == 0):
        camber = np.zeros(len(x))
    else:
        for i in x:
            if (i<p):
                camber = np.append(camber,((m/(p**2)) * (2*p*i - i**2)))
            else:
                camber = np.append(camber,((m/((1-p)**2)) * ((1 - (2*p)) + 2*p*i - i**2)))
    i = 0
    for j in x:
        upper = np.append(upper,camber[i] + (t/0.2)*(0.2969*(j)**(1/2) - 0.1260*j - 0.3516*j**2 + 0.2843*j**3 -
        lower = np.append(lower,camber[i] - (t/0.2)*(0.2969*(j)**(1/2) - 0.1260*j - 0.3516*j**2 + 0.2843*j**3 -
        i += 1

    z = np.concatenate((upper,lower))
    x = np.tile(x, 2)
    return x, z

'''
x14, z14 = naca4(0,0,14)
x12, z12 = naca4(0,0,12)
x2414, z2414 = naca4(2,4,12)
'''

folder = "airfoil-data"

filename = "naca0012.txt"
filename = os.path.join(folder,filename)
x12, z12 = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)

filename = "naca2414.txt"
filename = os.path.join(folder,filename)
x2414, z2414 = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)

filename = "naca0014.txt"
filename = os.path.join(folder,filename)
x14, z14 = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)

filename = "naca0012CpInvisc.txt"
filename = os.path.join(folder,filename)
x12_cp, cp12i = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)

filename = "naca0012CpVisc.txt"
filename = os.path.join(folder,filename)
x12_cp, cp12v = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)

filename = "naca2414CpInvisc.txt"
filename = os.path.join(folder,filename)
x2414_cp, cp2414i = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)

filename = "naca2414CpVisc.txt"
filename = os.path.join(folder,filename)
x2414_cp, cp2414v = np.loadtxt(filename, dtype=float, skiprows = 1, unpack = True)


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, based on airfoil geometry
    x_center = (x.max() + x.min()) / 2          # X coordinate of center of circle

    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
```

```python
    x_ends = np.copy(x_circle)                  # projection of the x-coord on the surface
    y_ends = np.empty_like(x_ends)              # initialization of the y-coord Numpy array

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

    # 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] <= x[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

rho = 1.6476e-3 #slugs
viscosity = 3.493e-7
Re = u_inf*1*rho/viscosity
print("Reynolds number: {}".format(Re))

n0012_panels = define_panels(x12,z12,102)
n2414_panels = define_panels(x2414,z2414,102)

analyze_panels(n0012_panels,u_inf)
analyze_panels(n2414_panels,u_inf)

print("")

'''
plt.figure()
plt.plot(np.append([panel.xa for panel in n0012_panels], n0012_panels[0].xa),
            np.append([panel.ya for panel in n0012_panels], n0012_panels[0].ya),
            linestyle='-', linewidth=1, marker='o', markersize=6, color='#CD2305')
'''

plt.figure(figsize=(10,6))
plt.gca().invert_yaxis()
plt.title("NACA 0012")
plt.xlabel(r"$\frac{x}{c}$")
plt.ylabel(r"$C_p$")
plt.plot([p.xc for p in n0012_panels], [p.cp for p in n0012_panels],
         color=colors[0],linewidth = 3,linestyle = "-", label = "Panel Method")
plt.plot(x12_cp,cp12i,color = colors[1],linestyle = "-", label = "Inviscid")
plt.plot(x12_cp,cp12v,color = colors[2],linestyle = "-", label = "Viscid with Re = {:.3g}".format(Re))
plt.legend()

plt.figure(figsize=(10,6))
plt.gca().invert_yaxis()
plt.title("NACA 2414")
plt.xlabel(r"$\frac{x}{c}$")
plt.ylabel(r"$C_p$")
plt.plot([p.xc for p in n2414_panels], [p.cp for p in n2414_panels],
         color=colors[0],linewidth = 3,linestyle = "-",label = "Panel Method")
plt.plot(x2414_cp,cp2414i,color = colors[1],linestyle = "-", label = "Inviscid")
plt.plot(x2414_cp,cp2414v,color = colors[2],linestyle = "-", label = "Viscid with Re = {:.3g}".format(Re))
plt.legend();
```

```
Reynolds number: 77356.54165473803
Panel Analysis for 102 Panels Complete!
```


NACA 0012
−0.25
0.00
0.25
0.50
0.75
1.00
C_p
0.0
0.2
0.4
0.6
0.8
1.0
$\frac{x}{c}$
Panel Method
Inviscid
Viscid with Re = 7.74e+04


NACA 2414
−0.5
0.0
0.5
1.0
C_p
0.0
0.2
0.4
0.6
0.8
1.0
$\frac{x}{c}$
Panel Method
Inviscid
Viscid with Re = 7.74e+04

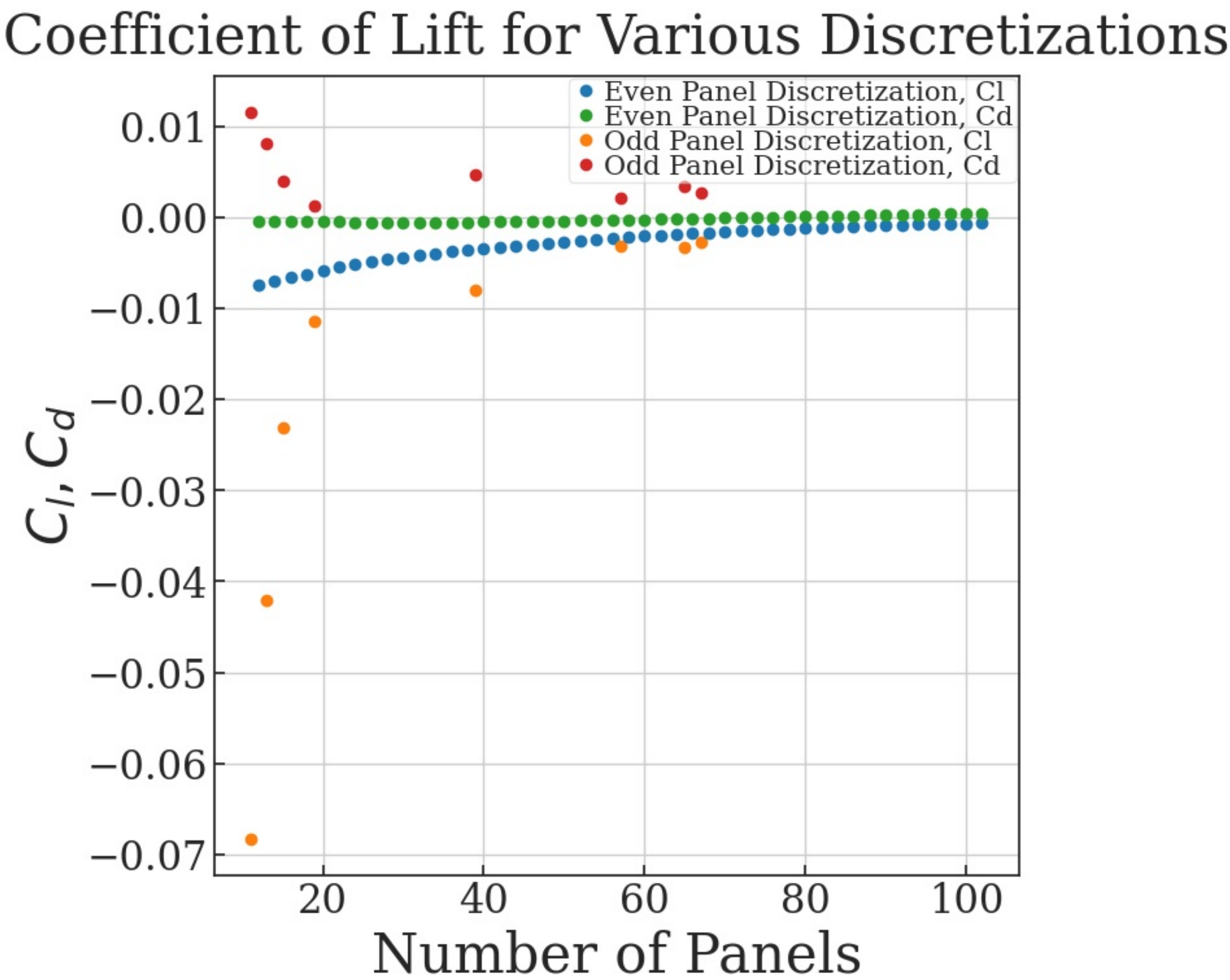
The accuracy of the panel method approximation with 102 panels is very good for the symmetric airfoil (NACA 0012). It closesly matches the inviscid XFOIL calculation, albeit with a noticeable spike at towards the aft. The accuracy of the panel method with 102 panels for the cambered airfoil (NACA 2414) appears to be less accurate than XFOIL. This panel method does not accurately account for the lift of the cambered airfoil at an angle of attack of 0.

The reasoning for picking 102 panels will be explained in the next section, which calculates C_l and C_d for the NACA 0012 symmetric airfoil at an angle of attack of zero degrees, at various discretizations.

In [ ]:

```
#Calculate the the integral of the upper and lower surface pressures and subtract for Cl
def getCl(numPanels):
    n0012_panels = define_panels(x12,z12,numPanels)
    analyze_panels(n0012_panels,u_inf)

    p_method_upper = np.empty((1,0))
    p_method_upperX = np.empty((1,0))
    p_method_upperCenterY = np.empty((1,0))
    p_method_lower = np.empty((1,0))
    p_method_lowerX = np.empty((1,0))
    p_method_lowerCenterY = np.empty((1,0))

    for p in n0012_panels:
        if p.loc == "upper":
            p_method_upper = np.append(p_method_upper,p.cp)
            p_method_upperX = np.append(p_method_upperX,p.xc)
            p_method_upperCenterY = np.append(p_method_upperCenterY,p.yc)
        else:
            p_method_lower = np.append(p_method_lower,p.cp)
            p_method_lowerX = np.append(p_method_lowerX,p.xc)
            p_method_lowerCenterY = np.append(p_method_lowerCenterY,p.yc)

    CnUpperInt = np.trapz(p_method_upper,p_method_upperX)
    CnLowerInt = -1 * np.trapz(p_method_lower,p_method_lowerX)

    Cl = CnLowerInt - CnUpperInt

    CaUpperInt = np.trapz(np.multiply(p_method_upper,np.gradient(p_method_upperCenterY,p_method_upperX)),p_meth
    CaLowerInt = np.trapz(np.multiply(p_method_lower,np.gradient(p_method_lowerCenterY,p_method_lowerX)),p_meth

    Cd = CaUpperInt - CaLowerInt

    return Cl, Cd

#plot Cl for various discretizations starting at 11 panels.
#odd panel numbers are in orange
plt.figure()
for i in range(11,103,1):
    if i%2 == 1:
        plt.scatter(i,getCl(i)[0],color=colors[1]) #print points if the panel is odd
        plt.scatter(i,getCl(i)[1],color=colors[3]) #print points if the panel is odd
    else:
        plt.scatter(i,getCl(i)[0],color=colors[0])
        plt.scatter(i,getCl(i)[1],color=colors[2])
print("")
plt.title("Coefficient of Lift and Drag for Various Discretizations")
plt.xlabel("Number of Panels")
plt.ylabel(r"$C_l, C_d$")

evenDotL = mlines.Line2D([], [], color=colors[0], linestyle='-', linewidth=0, marker = 'o', label='Even Panel D
oddDotL = mlines.Line2D([], [], color=colors[1], linestyle='-', linewidth=0, marker = 'o', label='Odd Panel Dis
evenDotD = mlines.Line2D([], [], color=colors[2], linestyle='-', linewidth=0, marker = 'o', label='Even Panel D
oddDotD = mlines.Line2D([], [], color=colors[3], linestyle='-', linewidth=0, marker = 'o', label='Odd Panel Dis

plt.legend(handles = [evenDotL,evenDotD,oddDotL,oddDotD],loc = "best");

#print("Panel Method Cl: {}".format(Cl))
#print("Xfoil Cl: ")
```

```
Panel Analysis for 102 Panels Complete!
```



These calculations reveal several very interesting aspects about the panel method and of this code. Firstly, as the number of panels used in our discretization ("panel number") gets larger, the approximation approaches a C_l of 0. This is the expected value, as a symmetric airfoil at zero angle of attack should not have any lift. Since the even panel numbers will cause the geometry to also be symmetric (almost), even panel numbers will approach 0 C_l much faster than odd panel numbers. This is clearly demonstrated in the graph, as the error of odd panel numbers is drastically higher than even panel numbers.

Similarly, for C_d, even panel numbers produced a very accurate result (all even panel numbers are very close to 0) and odd panel numbers produced a less accurate result. As was demonstrated in a previous project, C_d should be zero for a potential flow over a symmetric airfoil with an angle of attack of zero.

The most interesting (and also baffling) part of these results to me are that only a select few odd panel numbers create valid panel attributes. After many hours, I was able to find that the problem traces back to the tangential velocity integral, which somehow returns "NaN" for certain panels in certain odd panel distributions. 101 panels was one such distribution, which is why I chose to use 102 panels, even though the problem statement asked for 101 panels.