Potential Flow

Nicholas Richmond

Student ID: 919578115

Project #: 2

Date: 11/3/24

EAE 127: Applied Aerodynamics

University of California, Davis

In [295…

```
#standard imports and setups
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
### 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: Superposition of Elementary Flows

1.1: Superposition Plot

The first problem is about plotting superposition of flows. The elementary flows we will deal with are *freestream flow*, *flow sources*, and *flow sinks*. Each plot will also contain *streamlines*, the *dividing streamline*, and the locations of *sources*, *sinks*, and the *stagnation point*.

The following code simluates a doublet

In [567…

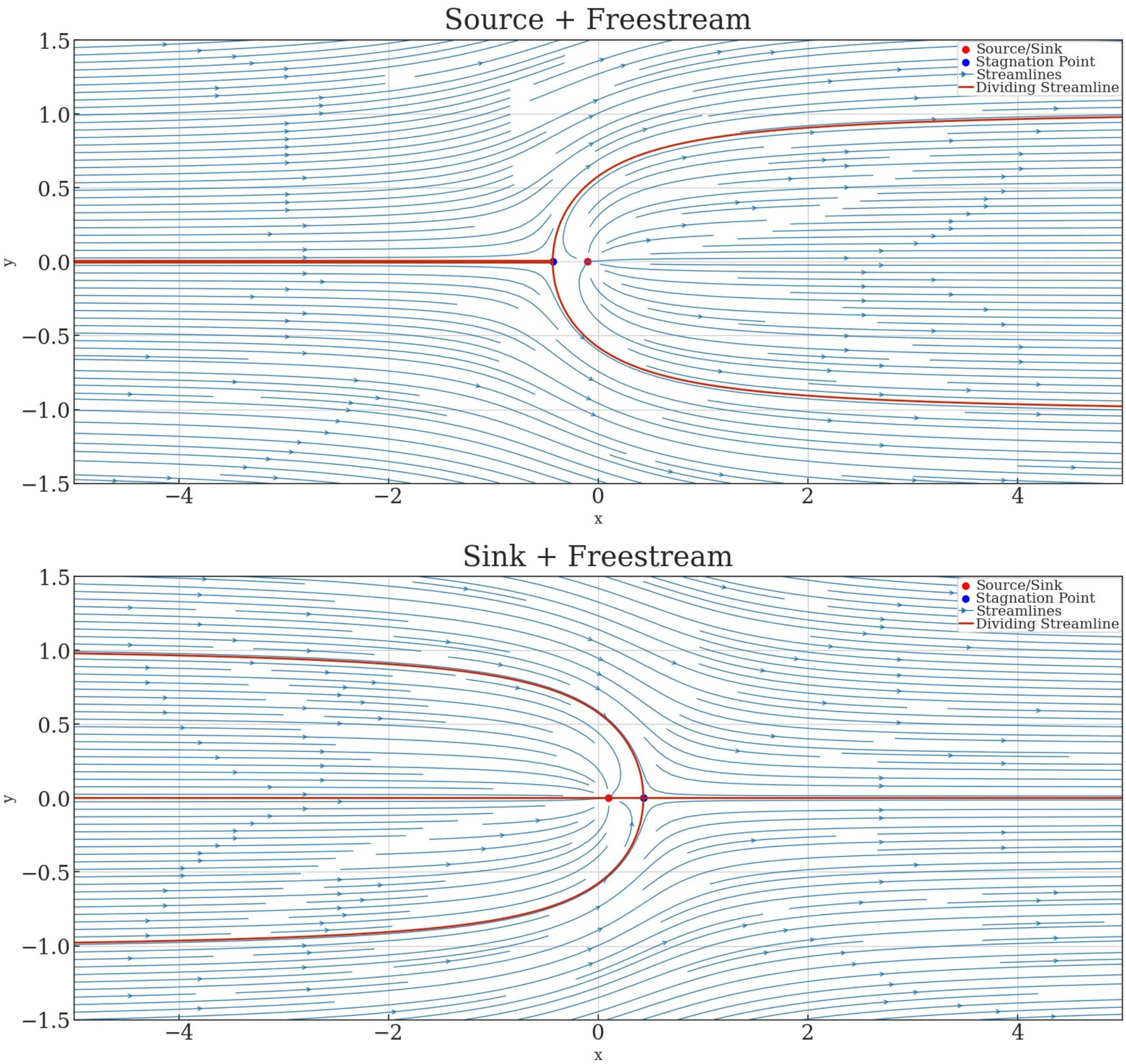
```
#Plot various flows using the superposition of a freestream flow, a source, and a sink

N = 250                                         # Number of points/sections to use in each direction for our fl
x_start, x_end = -5.0, 5.0                      # Boundaries of our flow in the x direction
y_start, y_end = -1.5, 1.5                      # Boundaries of our flow in the y direction

# Note, you can adjust the start and end points later to get the best image of your plot / flow

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

u_inf = 1.2                                    # Freestream flow velocity

# Computing the freestream velocity field
u_freestream = u_inf * np.ones((N, N), dtype=float)
v_freestream = np.zeros((N, N), dtype=float)

# Computing the stream-function
psi_freestream = u_inf * Y


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 get_stream_function(strength, xs, ys, X, Y):
    """
    Returns the stream-function generated by a source/sink.

    Parameters
    ----------
    strength: float
        Strength of the source/sink.
    xs: float
```

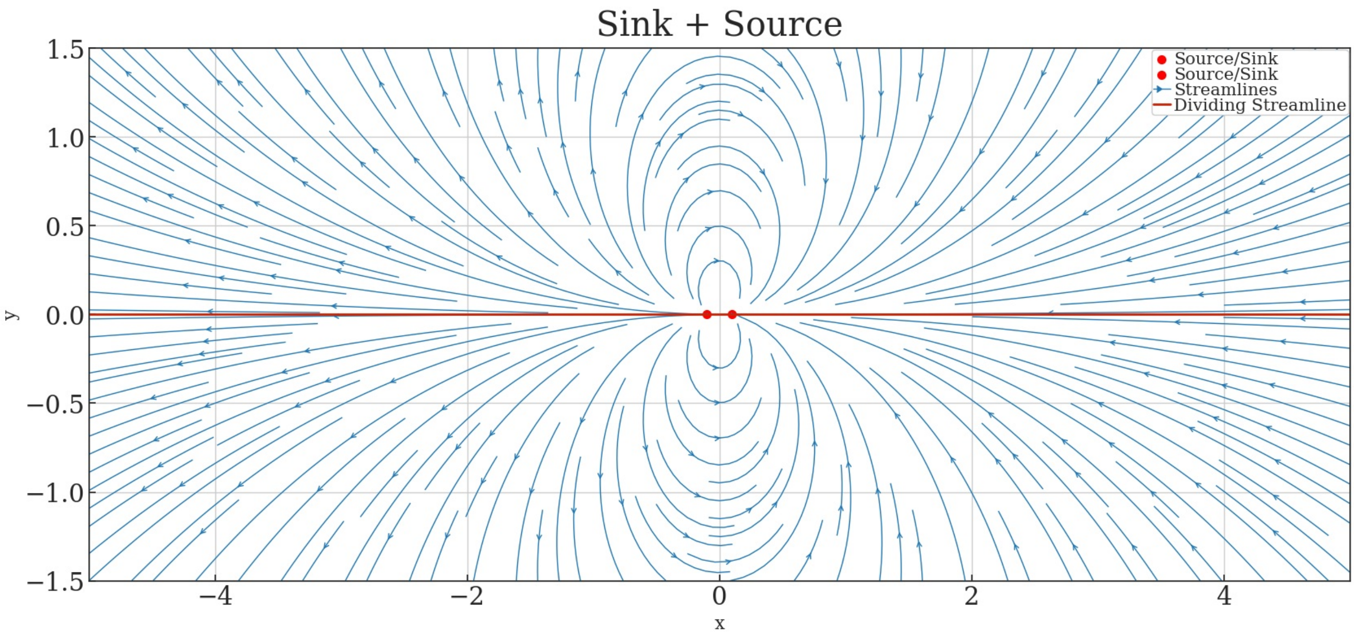
```python
        x-coordinate of the source (or sink).
    ys: float
        y-coordinate of the source (or sink).
    X: 2D Numpy array of floats
        x-coordinate of the mesh points.
    Y: 2D Numpy array of floats
        y-coordinate of the mesh points.

    Returns
    -------
    psi: 2D Numpy array of floats
        The stream-function.
    """
    # Here, input the equation for psi from the equations above
    psi = strength / (2 * np.pi) * np.arctan2((Y - ys), (X - xs))

    return psi

def get_velocity_doublet(Kappa, xs, ys, X, Y):
    """
    Returns the velocity field generated by a source/sink.

    Parameters
    ----------
    Kappa: float
        Strength of the doublet
    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 = Kappa  * (((X - xs)**2 - (Y - ys)**2))/ (2*np.pi *((X - xs)**2 + (Y - ys)**2)**2)

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

    return u, v

def get_stream_function_doublet(Kappa, xs, ys, X, Y):
    """
    Returns the stream-function generated by a source/sink.

    Parameters
    ----------
    Kappa: float
        Strength of the doublet.
    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
    -------
    psi: 2D Numpy array of floats
        The stream-function.
    """
    # Here, input the equation for psi from the equations above
    psi = Kappa * ((Y - ys)) / (2 * np.pi) * ((X - xs)**2 + (Y - ys)**2)

    return psi


source_strength =  2.5                          # Strength of source singularity
x_source = -0.1                                 # X coordinate of source singularity
y_source =  0.0                                 # Y coordinate of source singularity

# Here, we use some simple code to set up information about our source.
```

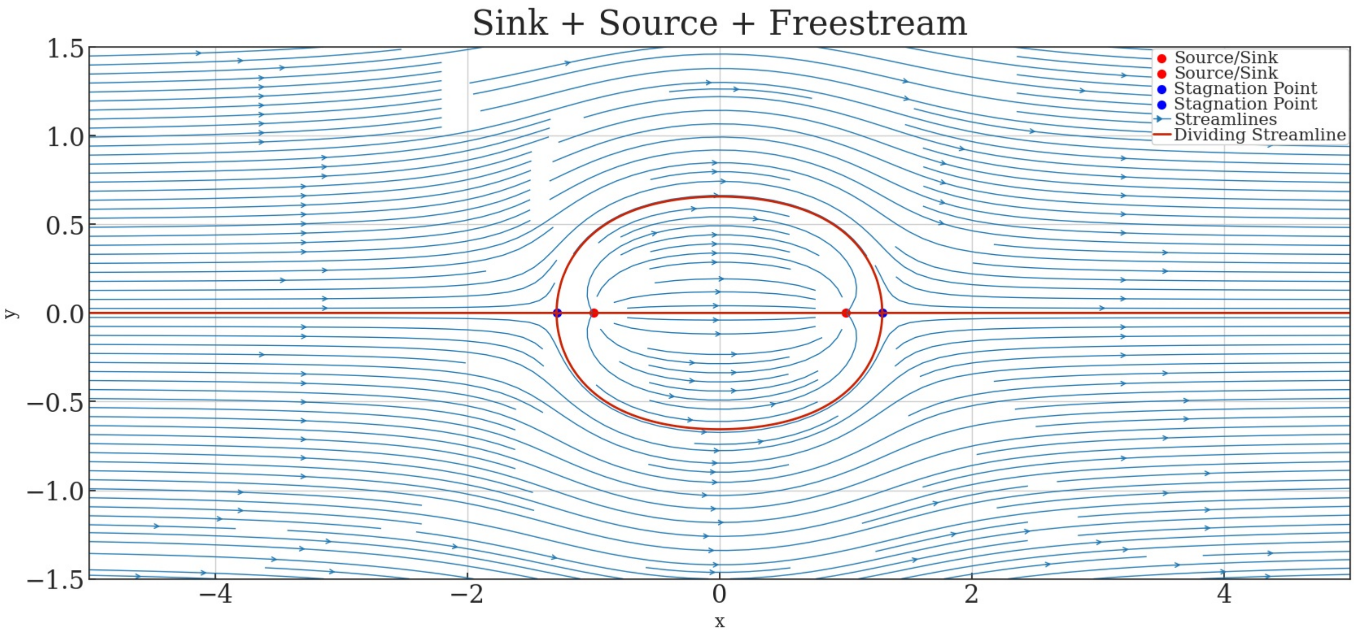
```
# Next we use the functions above to create the source and sink contributions to u, v and psi.

u_source, v_source = get_velocity(source_strength, x_source, y_source, X, Y)
psi_source = get_stream_function(source_strength, x_source, y_source, X, Y)

sink_strength = -1*source_strength
x_sink = 0.10

u_sink, v_sink = get_velocity(sink_strength, x_sink, y_source, X, Y)
psi_sink = get_stream_function(sink_strength, x_sink, y_source, X, Y)

# As we can see, the inputs for both functions are the same.

            #                          #
            # ***STAGNATION POINTS*** #
            #                          #


# Theory for where the stagnation point should be for a freestream flow + source + sink, assuming they are symme


x_stag_ssf_1 = 1*np.sqrt(1 + (source_strength)/(np.pi*u_inf))
x_stag_ssf_2 = -1*x_stag_ssf_1

# Stagnation points for a source in a freestream:
x_stag_source_freestream = -1* source_strength/(2*np.pi*u_inf) + x_source

# Stagnation points for a sink in a freestream:
x_stag_sink_freestream = -1*sink_strength/(2*np.pi*u_inf) + x_sink


            #                          #
            # ***STAGNATION POINTS*** #
            #                          #

width = 20
height = 8.5
# Setting up standard plotting functions once again


def plotScenario(U_Total,V_Total,psi_Total,stagLoc,sourceLoc,name):

    #plot the combined flows

    # The "total" variables now account for the contributions to the flow from both the source and the freestre
    # We can now plot this the same way we have above.

    plt.figure(figsize=(width, height))

    plt.title(name)
    plt.grid(True)
    plt.xlabel('x', fontsize=16)
    plt.ylabel('y', fontsize=16)
    plt.xlim(x_start, x_end)
    plt.ylim(y_start, y_end)

    # Streamplot is a new plotting function that we will use to display the streamlines.
    # Notice that when plotting the freestream flow, now we should use U_Total and V_Total
    plt.streamplot(X, Y, U_Total, V_Total, density=2,linewidth=1, arrowsize=1, arrowstyle='->')

    # A new plotting function we will use is contour(), this creates a contour of the flow.
    # We call this contour the *dividing streamline*. This requires an accurate stream function psi.
    if name == "Source + Freestream":
        plt.contour(X, Y, psi_Total, levels=[source_strength/2], colors='#CD2305', linewidths=2, linestyles='so
        plt.contour(X, Y, psi_Total, levels=[-1*source_strength/2], colors='#CD2305', linewidths=2, linestyles=
    else:
        plt.contour(X, Y, psi_Total, levels=[0], colors='#CD2305', linewidths=2, linestyles='solid');


    # We should also a point to the plot to represent the location of the source.
    # Remember that we should do this for every source and sink that we add, and we should make sure to
    # differentiate between sources and sinks when we have both.
    if sourceLoc != 0:
        for location in sourceLoc:
            plt.scatter(location, y_source, color = 'red', s = 50, marker ='o',label = "Source/Sink")

    if stagLoc != 0:
        for location in stagLoc:
            plt.scatter(location,0, color = "blue", s = 50, marker = 'o',label = "Stagnation Point")
    #plt.scatter(x_stag_2, y_source, color = "blue", s = 50, marker = 'o')

    plt.plot([5,5.01],[5,5.01],label = "Streamlines",color = colors[0],linewidth = 1, marker = 5) #add a line o
    plt.plot([5,5.01],[5,5.01],label = "Dividing Streamline",color = '#CD2305',linewidth = 2)
```

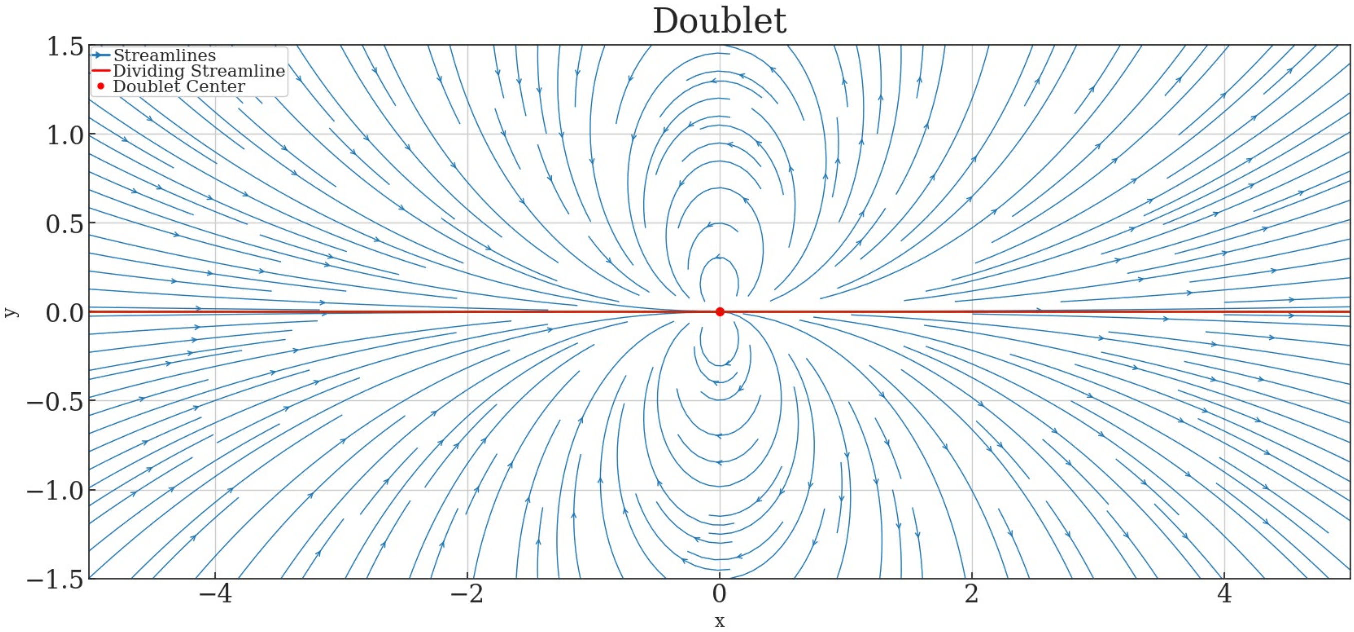
```python
    plt.legend(framealpha = 1,loc = "upper right")

#plot a source + freestream
U_Total = u_freestream + u_source
V_Total = v_freestream + v_source
psi_Total = psi_freestream + psi_source
sLocation = [x_source]
stagLoc = [x_stag_source_freestream]
plotScenario(U_Total,V_Total,psi_Total,stagLoc,sLocation,"Source + Freestream")

#plot a sink + freestream
U_Total = u_freestream + u_sink
V_Total = v_freestream + v_sink
psi_Total = psi_freestream + psi_sink
sLocation = [x_sink]
stagLoc = [x_stag_sink_freestream]
plotScenario(U_Total,V_Total,psi_Total,stagLoc,sLocation,"Sink + Freestream")

#plot a source and a sink near each other
U_Total = u_sink + u_source
V_Total = v_sink + v_source
psi_Total = psi_sink + psi_source
sLocation = [x_source,x_sink]
stagLoc = 0
plotScenario(U_Total,V_Total,psi_Total,stagLoc,sLocation,"Sink + Source")

#plot a source and a sink near each other in a freestream
x_sink = 1
x_source = -1

u_source, v_source = get_velocity(source_strength,x_source,0,X,Y)
u_sink, v_sink = get_velocity(sink_strength,x_sink,0,X,Y)
psi_source = get_stream_function(source_strength,x_source,0,X,Y)
psi_sink = get_stream_function(sink_strength,x_sink,0,X,Y)

U_Total = u_freestream + u_sink + u_source
V_Total = v_freestream + v_sink + v_source
psi_Total = psi_freestream + psi_sink + psi_source
sLocation = [x_source,x_sink]
stagLoc = [x_stag_ssf_1,x_stag_ssf_2]
plotScenario(U_Total,V_Total,psi_Total,stagLoc,sLocation,"Sink + Source + Freestream")

#plot a doublet

Kappa = 500000
x_loc = 0.0
y_loc = 0.0

u_doub, v_doub =  get_velocity_doublet(Kappa, x_loc, y_loc, X, Y)
psi_doub =  get_stream_function_doublet(Kappa, x_loc, y_loc, X, Y)

U_Total = u_doub
V_Total = v_doub
psi_Total = psi_doub

# The "total" variables now account for the contributions to the flow from both the source and the freestream.
# We can now plot this the same way we have above.


width = 20
height = 8.5
# Setting up standard plotting functions once again
plt.figure(figsize=(width, height))

plt.grid(True)
plt.xlabel('x', fontsize=16)
plt.ylabel('y', fontsize=16)
plt.xlim(x_start, x_end)
plt.ylim(y_start, y_end)
plt.title("Doublet")

# Streamplot is a new plotting function that we will use to display the streamlines.
# Notice that when plotting the freestream flow, now we should use U_Total and V_Total
plt.streamplot(X, Y, U_Total, V_Total, density=2, linewidth=1, arrowsize=1, arrowstyle='->')

# A new plotting function we will use is contour(), this creates a contour of the flow.
# We call this contour the *dividing streamline*. This requires an accurate stream function psi.
plt.contour(X, Y, psi_Total, levels=[0], colors='#CD2305', linewidths=2, linestyles='solid');

plt.scatter(0,0, color = 'red', s = 50, marker ='o')

line1 = mlines.Line2D([], [], color=colors[0], linestyle='-', linewidth=2, marker = 5, label='Streamlines')
line2 = mlines.Line2D([], [], color='red', linestyle='-', linewidth=0,marker = 'o',markersize = 5, label='Doubl
```

```
line3 = mlines.Line2D([], [], color='red', linestyle='-', linewidth=2, label='Dividing Streamline')

plt.legend(handles = [line1,line3,line2],loc="upper left",framealpha = 1);

# We should also a point to the plot to represent the location of the source.
# Remember that we should do this for every source and sink that we add, and we should make sure to
# differentiate between sources and sinks when we have both.


plt.show()
```




Sink + Source
Source/Sink
Source/Sink
Streamlines
Dividing Streamline
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−4
−2
0
2
4
x


Sink + Source + Freestream
Source/Sink
Source/Sink
Stagnation Point
Stagnation Point
Streamlines
Dividing Streamline
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−4
−2
0
2
4
x



The five plots above are four examples of super positions of various elementary flows, with each elements labeled. The final plot is of a doublet, which is mathematically the shape of a source and a sink as the limit of the distance between them goes to zero, but increasing their strengths such that the strength of the doublet \Kappa dividied by the distance between the source and the sink is constant.

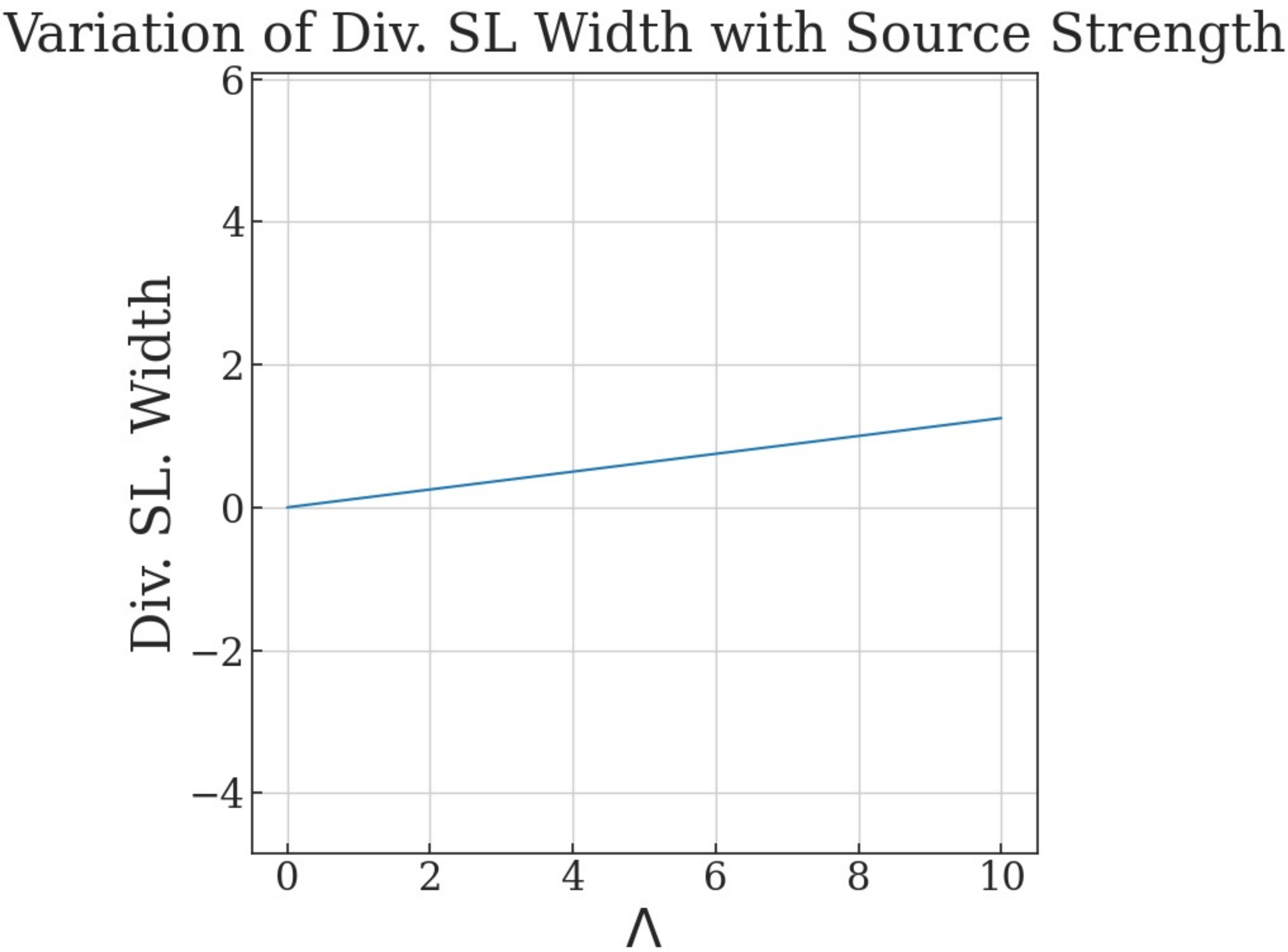
The final portion of part 1 asks to plot the dividing streamline diameter as a function of the source strength for the source in freestream flow. The diameter of the dividing streamline (or, more accurately, width) in this particular case can be found as:

$$Width = \frac{\Lambda}{V_\infty}$$

The derivation is left as an exercise to the grader. Plotting this width vs the source strength is shown in the next code segment, where $\Lambda \in [0, 10]\frac{m^2}{sec}$

In [589…

```
#Diameter vs source strength

Lambda = np.linspace(0,10,100)
Width = Lambda/u_inf

plt.figure(figsize=(8,8))
plt.title("Variation of Div. SL Width with Source Strength")
plt.xlabel(r"$\Lambda$")
plt.ylabel("Div. SL. Width")
plt.axis("equal")

plt.plot(Lambda,Width,color = colors[0]);
```



The relationship between the width of the dividing streamline and the strength of the source is linear, with a slope equal to the reciprocal of the freestream velocity. In other words, as the strength of the source increases, the width of the dividing streamline also increases.

Problem 2: Potential Flow Airfoil Representation

This problem asks to create a potential flow that approximates a NACA 0015 airfoil, and then finds the geometric error of the dividing streamline and the actual airfoil geometry

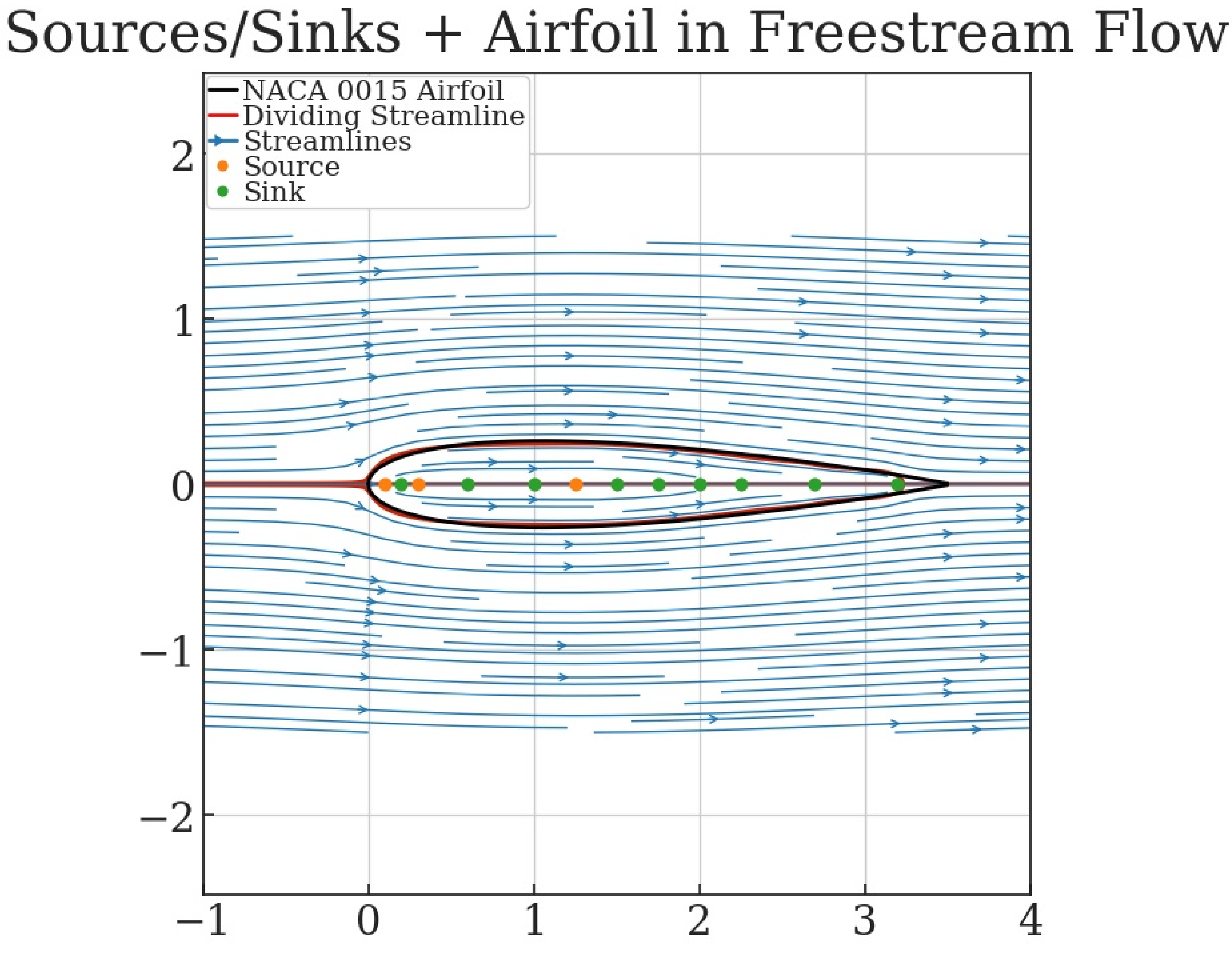
In [590…

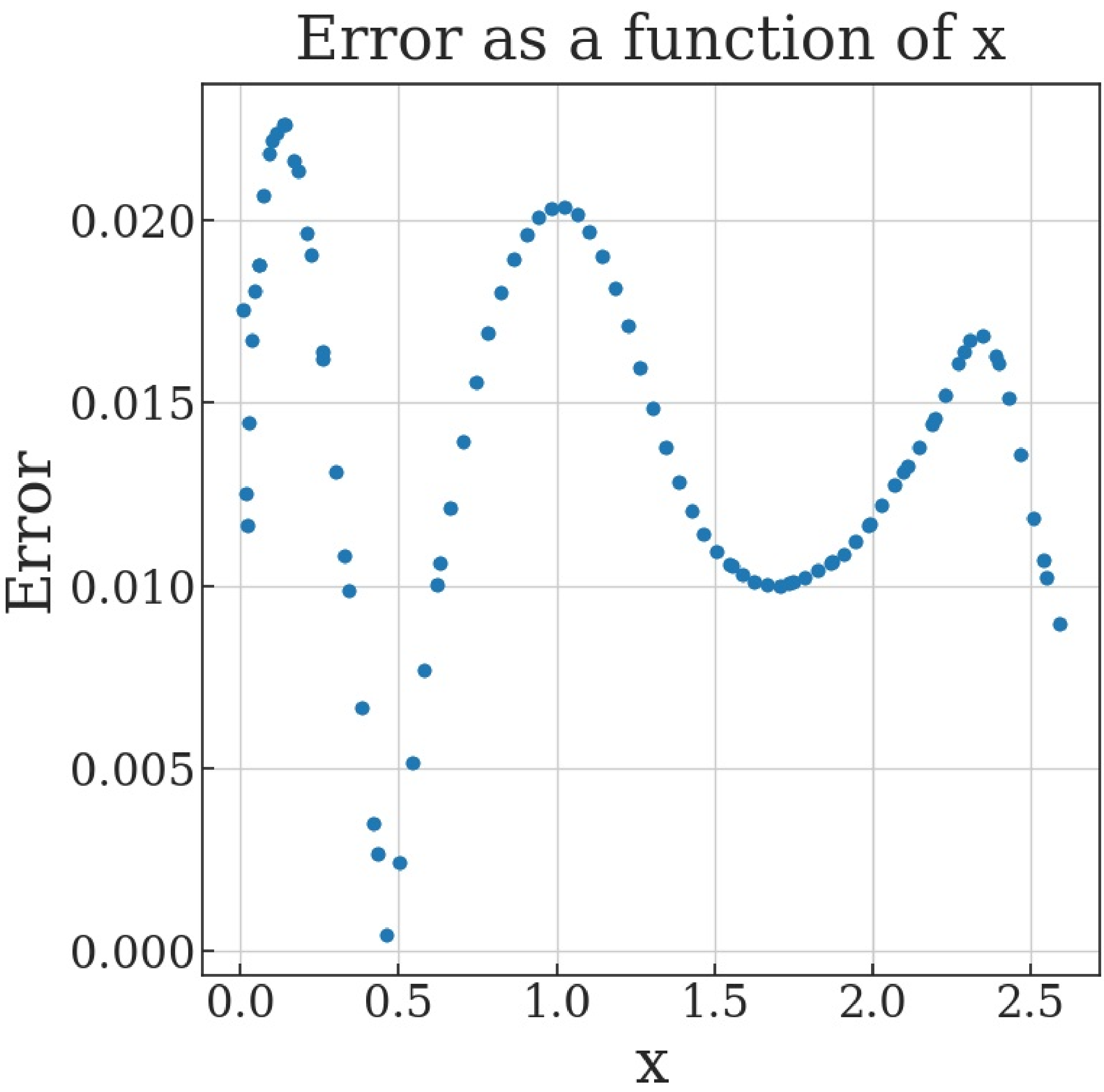
```
#Plot airfoil and potential approximation

#plot NACA 0015 airfoil the same way it was plotted in project 1, using the function already created:
stepsize = 0.00025
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
    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


    return upper,lower,camber, x

upperSurface, lowerSurface, camber, x_airfoil = naca4(0,0,15) #generate data for a NACA 0015 airfoil
```

```
U_Total = 0
V_Total = 0
psi_Total = 0

Strengths = np.array([4.8, -0.4, 0.1, -.1, -.1, 0.3, -.4, -0.5, -0.6, -0.8, -0.9, -1.5])
x_loc = np.array([0.1, 0.2, 0.3, 0.6, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.7, 3.2])
y_loc = np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

# The above arrays define the data/definitions to create 4 different sources/sinks.
# Lets see how we can write our for loop in order to loop through this data.

n = len(Strengths)                  # n is the number of sources/sinks we would like to create.

# As a reminder, a for loop repeats the code inside a certain number of times, in this case, we repeat the code

for i in range(0,n):              # This line basically reads for all values of i between 0 and n (in this case
    u, v = get_velocity(Strengths[i], x_loc[i], y_loc[i], X, Y)
    psi = get_stream_function(Strengths[i],x_loc[i], y_loc[i], X, Y)

    U_Total += u
    V_Total += v

    psi_Total += psi

u_inf = 8
u_freestream = u_inf * np.ones((N, N), dtype=float)
psi_freestream = u_inf * Y

U_Total += u_freestream
V_Total += v_freestream
psi_Total += psi_freestream

#rescale the x and z to the chord that was asked for
upperSurface = 3.5 * upperSurface
lowerSurface = 3.5 * lowerSurface
x_airfoil = x_airfoil * 3.5

plt.figure()
plt.title("Sources/Sinks + Airfoil in Freestream Flow")
plt.rc('legend',fontsize=15) #change legend size to accomodate large titles

plt.streamplot(X, Y, U_Total, V_Total, density=1.5,linewidth=1, arrowsize=1, arrowstyle='->')
plt.contour(X, Y, psi_Total, levels=[0], colors='#CD2305', linewidths=2, linestyles='solid',zorder = 1)

for i in range(len(x_loc)):
    s = x_loc[i]
    if Strengths[i] > 0:
        plt.scatter(s, y_source, color = colors[1], s = 40, marker ='o',zorder = 2)
    else:
        plt.scatter(s, y_source, color = colors[2], s = 40, marker ='o',zorder = 2)

plt.plot(x_airfoil,upperSurface,color = "black", linewidth = 2)
plt.plot(x_airfoil,lowerSurface,color = "black", linewidth = 2)
plt.axis("equal")
plt.xlim(-1,4)
plt.ylim(-1.5,1.5)

line1 = mlines.Line2D([], [], color='black', linestyle='-', linewidth=2, label='NACA 0015 Airfoil')
line2 = mlines.Line2D([], [], color='red', linestyle='-', linewidth=2, label='Dividing Streamline')
line3 = mlines.Line2D([], [], color=colors[0], linestyle='-', linewidth=2, marker = 5, label='Streamlines')
line4 = mlines.Line2D([], [], color=colors[1], linestyle='-', linewidth=0,marker = 'o',markersize = 5, label='S
line5 = mlines.Line2D([], [], color=colors[2], linestyle='-', linewidth=0,marker = 'o',markersize = 5, label='S

plt.legend(handles = [line1,line2,line3,line4,line5],loc="upper left",framealpha = 1);

# (below function from supplement)
# In order to extract the dividing streamline, we can use the following function.
def collect_contour(X, Y, psi, xs = 0, xe = 2.625, af = 1):
    if af == 1:
        levels = 0

    plt.figure()
    CS = plt.contour(X, Y, psi, levels=[0], colors='#CD2305', linewidths=2, linestyles='solid')
    P = CS.collections[0].get_paths()[0]
    plt.close()

    V = P.vertices
    x_coord = np.array(V[:,0])
    y_coord = np.array(V[:,1])


    # Now we would like to remove the coordinates that aren't relevant.
    # Since we are dealing with an airfoil, lets assume the airfoil begins at (0,0) for the leading edge.
```

```
    # You may need to adjust the valuye of xe depending on 3/4s of the chord.
    p = len(x_coord)

    for i in range(p-1, -1, -1):
        if (x_coord[i] < xs) or (x_coord[i] > xe):
            x_coord = np.delete(x_coord, i)
            y_coord = np.delete(y_coord, i)

    #remove the line going through the center of the airfoil
    p = len(x_coord)
    for i in range(p-1, -1, -1):
        if (abs(y_coord[i]) < 0.0001):
            x_coord = np.delete(x_coord, i)
            y_coord = np.delete(y_coord, i)

    y_coord = abs(y_coord)
    return x_coord, y_coord

# The above function will give the coordinates for your dividing streamline.
# From here you should interpolate these coordinates to a new vector for x and then use the formula provided
# In the problem statement to determine the error for each approximation.
# The only inputs you need will be X, Y, and the stream function for you airfoil approximation. You should chang
# levels to be equal to 0 in the plt.contour() function within the collect_contour function.

x_div, y_div = collect_contour(X, Y, psi_Total)


#plots the points to make sure I am not going crazy just yet
#plt.figure()
#plt.title("Plot of Countor Points")
#plt.scatter(x_div,y_div)
#plt.axis("equal")


error = 0
errorGraph = np.empty(0)
y_div = abs(y_div)
for i in range(len(x_div)):
    error += abs(y_div[i] - np.interp(x_div[i],x_airfoil,upperSurface))
    errorGraph = np.append(errorGraph,abs(y_div[i] - np.interp(x_div[i],x_airfoil,upperSurface)))

plt.figure()
plt.scatter(x_div,errorGraph)
plt.title("Error as a function of x")
plt.xlabel("x")
plt.ylabel("Error")
error *= 1/(len(x_div))
print("Arbitrary error function results (error up to the 3/4 chord point): {:.6f}".format(error))
```

```
Arbitrary error function results (error up to the 3/4 chord point): 0.014073
```


Sources/Sinks + Airfoil in Freestream Flow
NACA 0015 Airfoil
Dividing Streamline
Streamlines
Source
Sink
2
1
0
−1
−2
−1
0
1
2
3
4



The error calculated above is 0.014073. I don't have a good idea as to whether this is a good value or not, but my dividing streamline does seem to very closely match the airfoil geometry. I also have included a plot of the error at each point along the airfoil. This produced an trend that matches the dividing streamline to the line of the airfoil. The oscillating pattern of the error magnitude was also very interesting.

Problem 3: Additional Aerodynamics Problems

3.1: Similarity Parameters

We are given a jet flying at set parameters and are asked to provide wind tunnel parameters such that a 1/5 scale model of the jet will experience similar behavior. In order to do this, the Renolds numebrs and Mach numbers of both cases must match (dynamically similar flows).

Case 1, Real Jet: Lear Jet at $10\ km$ altitude, $200\ \frac{m}{s}$ flight velocity, where the density and temperature of the air are $0.414\ \frac{kg}{m^3}$ and $223\ K$.

Case 2: Wind Tunnel: 1/5 scale model of the Case 1 jet with a pressure of $1\ atm$.

The Reynold's Number Re and Mach Number Ma are given as the following relationships:

$$Re = \frac{\rho V L}{\mu}$$

$$Ma = \frac{V}{\sqrt{kRT}}$$

Assuming that k and R are constant for air at both altitudes, temperature and velocity are the only thing affecting Ma. Velocity is included in both Re and in Ma, so both will be needed to solve for all three parameters asked for in the problem.

Finally, we will assume that the viscosity of the air in the wind tunnel is related to temperature in accordance with the following formula:

$$\frac{\mu_1}{\mu_2} = \left(\frac{T_1}{T_2}\right)^{0.7}$$

The following code solves for the conditions of Case 1.

In [591…

```
v1 = 200# m/s
rho1 = 0.414 #kg/m^3
mu1 = 1.458e-5 #Ns/m^3
T1 = 223 #K
C1 = 1 #chord length

v2 = 0
rho2 = 0
mu2 = 1.789e-5 # Ns/m^3
T2 = 0
C2 = C1/5 #1/5 scale model, so the chord length is 1/5 of C1

k = 1.4
R = 287

Re1 = rho1*v1*C1/mu1
Ma1 = v1/np.sqrt(k*R*T1)

print("|------------------------------------|")
print("  Reynods Number for Case 1: {:.4g}\n  Mach Number for Case 1: {:.4g}".format(Re1,Ma1))
print("|------------------------------------|")
```

```
|------------------------------------|
  Reynods Number for Case 1: 5.679e+06
  Mach Number for Case 1: 0.6681
|------------------------------------|
```

These two values need to be matched exactly for coefficient of lift and drag of the scale model to match the full scale aircraft.

{Re1\over Re2} = 1 = {{\rho_1V_1\over\mu_1}\over{\rho_2V_2\over 5\mu_2}} \longrightarrow \rho_2V_2\left({223\over T_2}\right)^{0.7} = 414 \newline\space\newline {Ma_1\over Ma_2} = 1 = {0.6681\over{V_2\over \sqrt{(1.4)(287)T_2}}}\longrightarrow {V_2\over \sqrt{T_2}} = 13.39

The atmospheric pressure in the wind tunnel is known at $1\ atm$. This gives us the final equation to be able to solve for each of the three variables using the ideal gas law.

$$P = \rho RT \longrightarrow \rho_2 T_2 = \frac{P}{R} = \frac{101000}{287} \longrightarrow \rho_2 T_2 = 351.92$$

With these three equations, we can get python to solve for each variable.

In [602…

```
from scipy.optimize import fsolve


# Define the system of equations
def system(vars):
    v_2,rho_2,T_2 = vars
    eq1 = rho_2*v_2*(223/T_2)**0.7 - 414  # Equation with sqrt(x)
    eq2 = v_2/(T_2)**(1/2) - 13.39        # Equation with sqrt(y)
    eq3 = rho_2*T_2 - 351.92               # Equation with sqrt(z)
    return [eq1, eq2, eq3]

# Initial guesses for x, y, and z (choose non-negative guesses to satisfy sqrt constraints)
initial_guesses = [200, 1, 200]

# Solve the system
solution = fsolve(system, initial_guesses, xtol=0.00000000001)

v2 = solution[0]
rho2 = solution[1]
T2 = solution[2]

print(f"Velocity = {solution[0]:.5g}")
print(f"Density = {solution[1]:.5g}")
print(f"Temperature = {solution[2]:.5g}")
```

```
Velocity = 178.56
Density = 1.9788
Temperature = 177.84
```

Note: The syntax used was derived with the help of Chat GPT and slightly edited by me.

After running the solver, these are the final values:

\rho_2 = 1.979 \space {kg\over m^3}\newline \space\newline V_2 = 178.56 \space {m\over s}\newline\space\newline T_2 = 177.84 \space K

These values can be verified to ensure $Re_1 = Re_2$ and $Ma_1 = Ma_2$

In [628…

```
#Verify the new values cause the same reynolds number and mach number

Re2 = rho2*v2*(1/5)/(mu1*(T2/223)**0.7)
Ma2 = v2/np.sqrt(k*R*T2)

print(f"Reynolds Number Case 1: {Re1:.5g}\nReynolds Number Scale Model: {Re2:.5g}")
print(f"Mach Number Case 1: {Ma1:.3g}\nMach Number Scale Model: {Ma2:.3g}")
```

```
Reynolds Number Case 1: 5.679e+06
Reynolds Number Scale Model: 5.679e+06
Mach Number Case 1: 0.668
Mach Number Scale Model: 0.668
```

These air parameters are within the realm of possibility, however the density is rather high for air and the temperature is rather low. This could be related to the assumptions made for the viscosity, which I took from Fluid Mechanics by Frank M. White, or it could be an error introduced from the tables used for the standard atmosphere. Regardless, these conditions do result in the same reynolds number and mach number for the scale model and the jet.

3.2: Lift Coefficient

The final part of this problem asks us to esimate the coefficient of lift for a Boing 787 at maximum gross weight while cruising at 42000 feet assuming standard atmosphere.

The formula for lift is:

$$L = \frac{1}{2}C_L\rho V^2 S$$

where S is the surface area of the wing, C_L is the coefficient of lift, ρ is the density of the air, and V is the airspeed. Since we are assuming level flight, lift must equal drag.

With a quick google search, a Boeing 787-9 has a wing surface area of 4058 square feet, has a cruising mach number of 0.85, and has a maximum takeoff weight of 254000 kilograms, or 560000 lbs. At 42000 feet, the density is $0.5315 \times 10^{-3} \frac{slugs}{ft^3}$ and the temperature is 216.67 K.

In [629…

```
#Solve for the coefficient of lift
W = 560000 #lbs
rho = 0.5315e-3 #slugs/ft^3
Ma = 0.85
k = 1.4
R = 287 #J/Kg K
T = 216.67 #kelvin
S = 4058 #ft^2

V = Ma*np.sqrt(k*R*T) # meters per second
V *= 3.2808 #convert to feet per second
CL = W*2/(rho*V**2*S)

print("|-----------------------------------|")
print(f"Calculated Coefficient of Lift: {CL:.5g}")
print("|-----------------------------------|")
```

```
|-----------------------------------|
Calculated Coefficient of Lift: 0.767
|-----------------------------------|
```

V = Ma\sqrt{kRT} = 0.85\sqrt{(1.4)(287)(216.67)}\longrightarrow V= 250.80 \space{m\over s} = 822.83 \space{ft\over s}\newline 560000 = {1\over2}C_

This coefficient of lift is in the right ballpark for a plane like this (somewhere between 0.5 and 1). Since coefficient of lift is dependent on the Reynolds number, our flight conditions as well as the parameters of the plane that I found from the internet will affect the coefficient of lift, and thus could introduce some variation.