Roll no	23M0009	**Name**	Ranjith A V
Department	Aerospace Engineering	**Program**	M.Tech. (Aerodynamics)

Academic Performance Summary

Year	Sem	SPI	CPI	Sem Credits Used for SPI	Completed Semester Credits	Cumulative Credits Used for CPI	Completed Cumulative Credits
2023	Spring	9.14	8.79	28.0	28.0	56.0	56.0
2023	Autumn	8.43	8.43	28.0	28.0	28.0	28.0

Semester-wise Details

*This registration is subject to approval(s) from faculty advisor/Course Instructor/Academic office.

Year/Semester: 2024-25/Autumn

Course Code	Course Name	Credits	Tag	Grade	Credit/Audit
AE 429	Aircraft Design Project	6.0	Department elective	Not allotted	C
CS 725	Foundations of Machine Learning	6.0	Core course	Not allotted	A
ME 782	Design Optimization	6.0	Additional Learning	Not allotted	C

Year/Semester: 2024-25/Project

Course Code	Course Name	Credits	Tag	Grade	Credit/Audit
AE 796	I Stage Project	42.0	Core course	Not allotted	C

Year/Semester: 2023-24/Spring

Course Code	Course Name	Credits	Tag	Grade	Credit/Audit
AE 650	Mini Project	6.0	Department elective	AA	C
AE 694	Seminar	4.0	Core course	AA	C
AE 706	Computational Fluid Dynamics	6.0	Core course	AB	C
AE 714	Aircraft Design	6.0	Department elective	AB	C

AE 899	Communication Skills	6.0	Core course	PP	N
EE 769	Introduction to Machine Learning	6.0	Core course	AU	A
ME 673	Mathematical Methods in Engineering	6.0	Additional Learning	CC	C
TD 656	Characterizing Hydro-Meteorological Hazards & Risk	6.0	Institute elective	BB	C

Year/Semester: 2023-24/Autumn

Course Code	Course Name	Credits	Tag	Grade	Credit/Audit
AE 611	Aerodynamics Lab	4.0	Core course	BB	C
AE 616	Gas Dynamics	6.0	Core course	AA	C
AE 623	Computing of Turbulent Flows	6.0	Additional Learning	AB	C
AE 705	Introduction to Flight	6.0	Core course	BB	C
AE 707	Aerodynamics of Aerospace Vehicles	6.0	Core course	BC	C
AE 725	Air Transportation	6.0	Department elective	AB	C
GC 101	Gender in the workplace	0.0	Core course	PP	N
TA 101	Teaching Assistant Skill Enhancement & Training (TASET)	0.0	Core course	PP	N

Report Problem

AE 707: Aerodynamics of Aerospace Vehicles

Developing Python Classes for the Vortex Lattice Method and Thin Airfoil Theory

Submitted by

23M0007 (*Vanhar Ali Shaik*)

23M0009 (*Ranjith A V*)

23M0035 (*Mahesh Bayas*).

The typical argument list of the function is **(SwpCQrtr,TprRatio,AspctRatio,NLtcX,NLtcY)**where,

SwpCQrtr: Sweep angle of the quarter-chord line (in degrees); positive (resp. negative) imply swept forward (resp. backward) wings

TprRatio: Taper ratio (ct/cr)

AspctRatio: Aspect ratio

NLtcX: No. of vortex lattices arrayed along longitudinal (i.e., x-axis) of wing NLtcY: No. of

vortex lattices arrayed along the half-span (i.e., y-axis) of wing

N.B.: We are not accounting for camber or wing twist since the program is specialized for planarwings.

Following Bertin textbook notations for bound vortex coordinates ((x1n, y1n),(x2n, y2n)),andcontrol points (xm, ym)

SwpCQrtr denotes quater chord sweep angletr

represents taper ratio

b represents span of wing

nLtcx and nLtcy represent number of divisions along longitudinal axis and lateral axis (along halfwing span) respectively

AR represents Aspect ratio

cr and ct represent root chord and tip chord length

c denotes spanwise chord length distribution from root to tipUinf denotes

freestream velocity

alpha denotes angle of attack S

denotes surface area of wing

input the wing geometry and freestream parameters sweptback if sweep angle (-)

sweptforward if sweep angle (+)

[471]:
```python
import numpy as np
import matplotlib.pyplot as plt

span = 1 #saying the value of span of wing is equal to 1,can be changed
 ↪according to the question.
# The typical argument list of the function VLM.
def VLM(SwpCQrtr, TprRatio, AspctRatio, NLtcX, NLtcY):
    SwpCQrtr=-1*SwpCQrtr #To account for -ve angle for swept back and +ve angle
 ↪for swept forward
    tr=TprRatio    #Taper ratio which is ct/cr.
    nLtcx=NLtcX   #No. of vortex lattices arrayed along longitudinal (i.e.,
 ↪x-axis) of wing
    nLtcy=NLtcY   #No. of vortex lattices arrayed along the half-span (i.e.,
 ↪y-axis) of wing
    AR=AspctRatio #represents Aspect ratio.
    b=span         #represents span of wing.
    cr=2*b/AR/(1+tr) #represent root chord length.
    ct=tr*cr   #represents tip chord length.
    rho=1.22  #denotes freestream density.
    Uinf=1    #denotes freestream velocity.
    alpha=10  #denotes angle of attack.

    S=((ct+cr)/2)*b   #Calculating area of the wing.
    print("The length of root chord: ",cr)
    print("The length of tip chord: ",ct)

# where delta is the swept angle of the wing leading edge
# The Leading edge sweep angle varies for delta (pure delta i.e taper ratio =0)
 ↪and general wing
    if tr == 0:
        delta= np.arctan(4*(1-tr)/AR/(1+tr))
    else:
        delta=np.arctan((cr-ct+2*b*np.tan(SwpCQrtr*np.pi/180))/(2*b))

# To store all the chord length values along the span from root to tip varying
 ↪with taper ratio
    c = np.zeros(nLtcy+1)
    for i in range(nLtcy+1):
        chord=cr-(i*(cr-ct)/nLtcy)
        c[i]=chord    # Distribution of chord length.
    print("The chord length distribution from root to tip: ",c)
    print("The sweep angle of wing leading edge: ",delta*180/np.pi, "\u00B0")
```

```python
# declaring the arrays to store all the coordinates of bound vortex and control
  ↪points
  x1n = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of root side of star board side of bound vortex .
  x2n = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of tip side of star board side of bound vortex .

  y1n = np.zeros((nLtcx, nLtcy))
  #to store y co-ordinates of root side of star board side of bound vortex .
  y2n = np.zeros((nLtcx, nLtcy))
  #to store y co-ordinates of tip side of star board side of bound vortex .

  xm = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of control points.
  ym = np.zeros((nLtcx, nLtcy))
  #to store y co-ordinates of control points

  Y1n = np.zeros((nLtcx, nLtcy))
  #to store y co-ordinates of root side of port board side of bound vortex.
  Y2n = np.zeros((nLtcx, nLtcy))
  #to store y co-ordinates of tip side of port board side of bound vortex .

  xl1 = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of root side of star board side of leading edge .
  xl2 = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of tip side of star board side of leading edge .

  yl1 = np.zeros((nLtcx, nLtcy))
  # to store y co-ordinates of root side of star board side of leading edge .
  yl2 = np.zeros((nLtcx, nLtcy))
  #to store y co-ordinates of tip side of star board side of leading edge .

  xt1 = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of root side of star board side of trailing edge .
  xt2 = np.zeros((nLtcx, nLtcy))
  #to store x co-ordinates of tip side of star board side of trailing edge .

#trailing edge y coordinates are same as leading edge y coordinates



# calculating the coordinates of bound vortex and control points for each panel
  ↪of NLtcX * NLtcY divisions
# Y1n and Y2n denote the span wise coordinates of bound vortex on port side of
  ↪wing
# x1n and x2n values remain same on port and star board side y1n and y2n values
  ↪will differ
  k=0                    # Considering the planar wing condition.
```

```
for i in range(nLtcx):
    for j in range(nLtcy):
        # calculating the x positions of the bound vortex
        x1n[i][j]=(j*np.tan(delta)*(b/2)/nLtcy) + (c[j]/(nLtcx*4)+i*c[j]/nLtcx)
        x2n[i][j]=((j+1)*np.tan(delta)*(b/2)/nLtcy) + (c[j+1]/(nLtcx*4)+i*c[j+1]/nLtcx)

        # calculating leading edge and trailing edge line coordinates
        xl1[i][j]=(j)*np.tan(delta)*(b/2)/nLtcy
        xl2[i][j]=(j+1)*np.tan(delta)*(b/2)/nLtcy
        xt1[i][j]=(j)*np.tan(delta)*(b/2)/nLtcy+c[j]
        xt2[i][j]=(j+1)*np.tan(delta)*(b/2)/nLtcy+c[j+1]

        # calculating the y positions of the bound vortex
        y1n[i][j]=j*(b/2)/nLtcy
        y2n[i][j]=(j+1)*(b/2)/nLtcy

        Y1n[i][j] = -1*y1n[i][j]
        Y2n[i][j] = -1*y2n[i][j]

        xm[i][j]=(xl1[i][j]+xl2[i][j])/2+((c[j]+c[j+1])/(2*nLtcx))*0.75+(i*(c[j]+c[j+1])/(2*nLtcx))
        ym[i][j]=((y1n[i][j]+y2n[i][j])/2)


# to print the coordinates of bound vortex, control points, leading edge and trailing edge
for i in range(nLtcx):
    for j in range(nLtcy):
        plt.plot([x1n[i][j], x2n[i][j]], [y1n[i][j], y2n[i][j]], marker='o', color='yellow')
        plt.plot([x1n[i][j], x2n[i][j]], [Y1n[i][j], Y2n[i][j]], marker='o', color='yellow')
        plt.plot(xm[i][j], ym[i][j],xm[i][j], -1*ym[i][j], marker='o', color='blue',markersize=2)
        plt.plot()
        plt.plot([xl1[i][j], xl2[i][j]], [y1n[i][j], y2n[i][j]], marker='o', color='red')
        plt.plot([xl1[i][j], xl2[i][j]], [Y1n[i][j], Y2n[i][j]], marker='o', color='red')
        plt.plot([xt1[i][j], xt2[i][j]], [y1n[i][j], y2n[i][j]], marker='o', color='red')
        plt.plot([xt1[i][j], xt2[i][j]], [Y1n[i][j], Y2n[i][j]], marker='o', color='red')
plt.plot([xl2[i][j], xt2[i][j]], [y2n[i][j], y2n[i][j]], marker='o', color='red')
```

```
  plt.plot([xl2[i][j], xt2[i][j]], [Y2n[i][j], Y2n[i][j]], marker='o',␣
↪color='red')
  plt.title('Panel representation of swept planar wing')
  plt.xlabel('longitudinal-axis (x)')
  plt.ylabel('span wise-axis (y)')
  plt.show()


# to calculate the downwash at panel 1 due to the contribution of panel 1,2,3...
↪(NLtcX * NLtcY).
# similarly calculating downwash at panel 2 due to the contribution of panel␣
↪1,2,3...(NLtcX * NLtcY).
# and also accounting for the contribution of panels of port side of the wing␣
↪on star board side.
# storing all the circulation coefficients in the matrix g
  g = []
  for i in range(nLtcx):
    for j  in range(nLtcy):
      m = 0
      ws = np.zeros((nLtcx*nLtcy))
      for k in range(nLtcx):
        for l in range(nLtcy):
          ws[m]=(1/
↪((xm[i][j]-x1n[k][l])*(ym[i][j]-y2n[k][l])-(xm[i][j]-x2n[k][l])*(ym[i][j]-y1n[k][l]))*(((x2
↪np.
↪sqrt((xm[i][j]-x1n[k][l])**2+(ym[i][j]-y1n[k][l])**2)-((x2n[k][l]-x1n[k][l])*(xm[i][j]-x2n[
↪np.sqrt((xm[i][j]-x2n[k][l])**2+(ym[i][j]-y2n[k][l])**2))+(1/
↪(y1n[k][l]-ym[i][j])*(1+(xm[i][j]-x1n[k][l])/np.
↪sqrt((xm[i][j]-x1n[k][l])**2+(ym[i][j]-y1n[k][l])**2))-(1/
↪(y2n[k][l]-ym[i][j])*(1+(xm[i][j]-x2n[k][l])/np.
↪sqrt((xm[i][j]-x2n[k][l])**2+(ym[i][j]-y2n[k][l])**2)))))-\
          (1/
↪((xm[i][j]-x1n[k][l])*(ym[i][j]-Y2n[k][l])-(xm[i][j]-x2n[k][l])*(ym[i][j]-Y1n[k][l]))*(((x2
↪np.
↪sqrt((xm[i][j]-x1n[k][l])**2+(ym[i][j]-Y1n[k][l])**2)-((x2n[k][l]-x1n[k][l])*(xm[i][j]-x2n[
↪np.sqrt((xm[i][j]-x2n[k][l])**2+(ym[i][j]-Y2n[k][l])**2))+(1/
↪(Y1n[k][l]-ym[i][j])*(1+(xm[i][j]-x1n[k][l])/np.
↪sqrt((xm[i][j]-x1n[k][l])**2+(ym[i][j]-Y1n[k][l])**2))-(1/
↪(Y2n[k][l]-ym[i][j])*(1+(xm[i][j]-x2n[k][l])/np.
↪sqrt((xm[i][j]-x2n[k][l])**2+(ym[i][j]-Y2n[k][l])**2)))))
          m = m + 1
      g.append(ws)
      ws = np.zeros((nLtcx*nLtcy))


# to calculate the inverse of the matrix g
```

```
  h  =  np.linalg.inv(g)

# calculating the circulation values by solving for X in [A][X]=[B] matrix

  gamma=[]
  for i in range(nLtcx):
    sum=0
    for j in range(nLtcy):
      sum=sum+h[j]          # transpose matrix (ht) not used
    gamma.append(sum)
# calculating total circulation
  total_gamma  =  -1*4*np.pi*b*Uinf*alpha*(np.pi/180)*np.sum(gamma)*(b/(2*nLtcy))

  Lift=2*rho*Uinf*total_gamma
  print("The Lift force is: ",Lift," N")

  Cl=Lift/(0.5*rho*(Uinf**2)*S)
  print("The Lift coefficient is: ",Cl, "at 10\u00B0 AOA")
  print("The Lift curve slope is: ",Cl/(alpha),"per degree")
  print("The Lift curve slope is: ",Cl/(alpha*np.pi/180), "per radian")

 ␣
 ↪print("\n*********************************************************************\n")
```

——————————————————————————-

Example 7.2

Quarter chord Sweep angle: 45°

Taper ratio: 1

Aspect ratio: 5

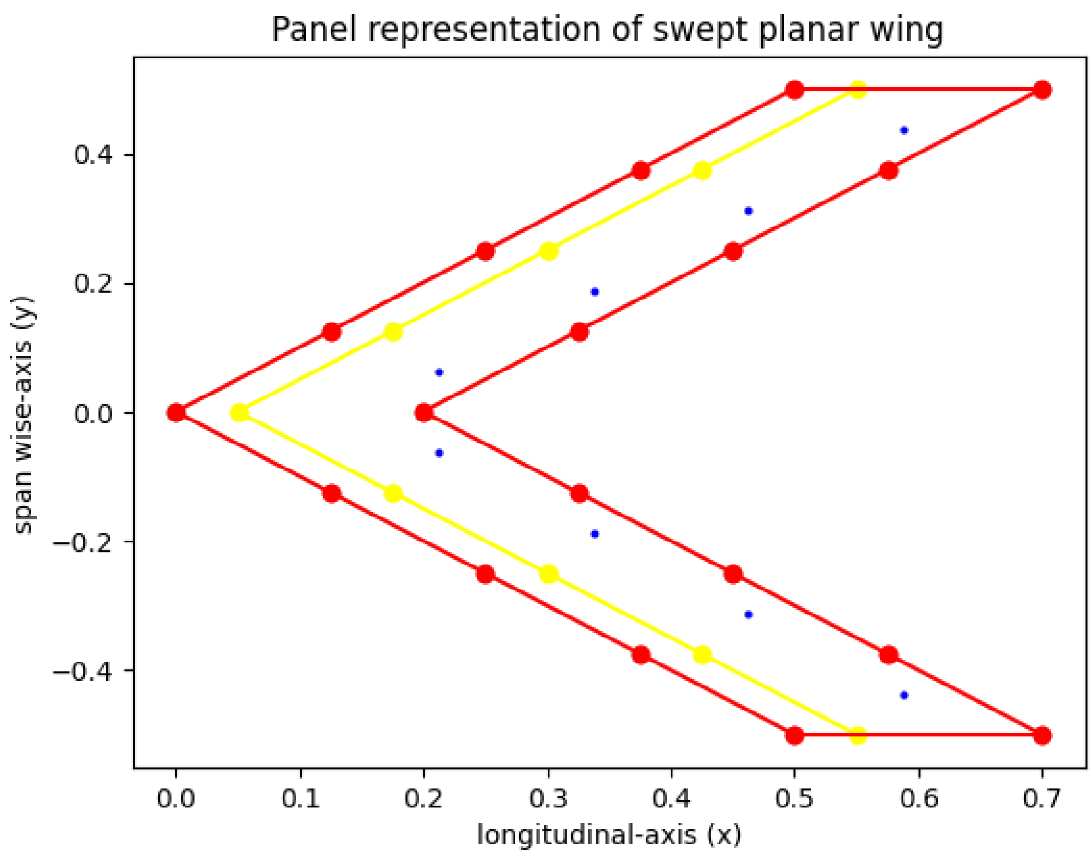
No. of divisions along chord (nLtcx): 1

No. of divisions along span (nLtcy): 4

[472]:
```
# VLM(SwpCQrtr, TprRatio, AspctRatio, NLtcX, NLtcY)
VLM(-45, 1, 5, 1, 4)

# Comparing results with experimental values
Slope = 3.444224187713715     # in per radians
Slope1 = Slope*np.pi/180      # in per degrees

x1  = np.linspace (0,12,400)
x2 = [2.0625, 4.1042, 6.2708, 8.3125, 10.2917]

# assuming the wing section as symmetric
y1  = Slope1*x1
```

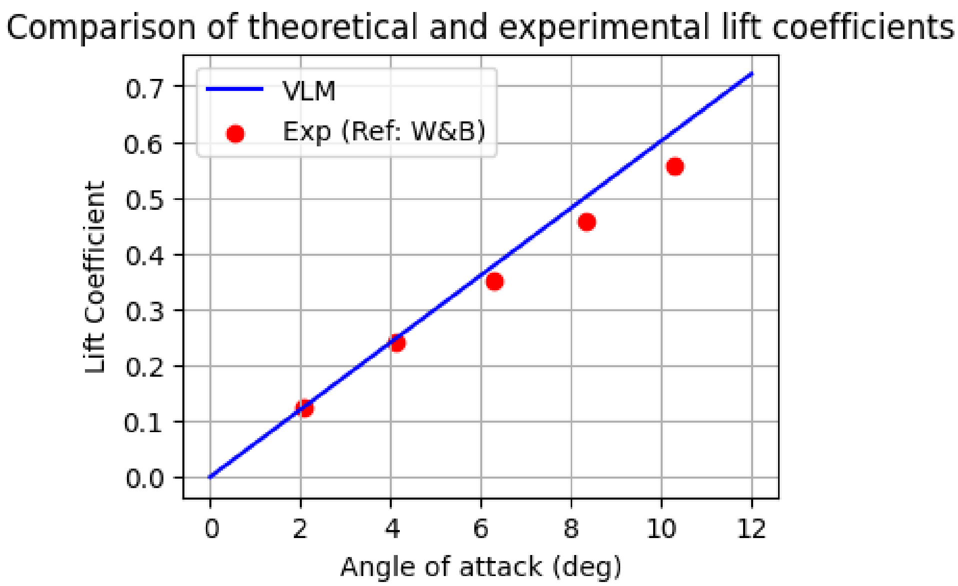
```
y2 = [0.1239, 0.2427, 0.3511, 0.4573, 0.5583]


plt.figure(figsize=(4,3))
plt.plot(x1,y1, color="blue", label = "VLM")
plt.scatter(x2,y2, color="red",marker="o", label = "Exp (Ref: W&B)")

plt.xlabel("Angle of attack (deg)")
plt.ylabel("Lift Coefficient")
plt.title("Comparison of theoretical and experimental lift coefficients")
plt.legend()
plt.grid(True)
plt.show()
```

The length of root chord: 0.2
The length of tip chord: 0.2
The chord length distribution from root to tip: [0.2 0.2 0.2 0.2 0.2]
The sweep angle of wing leading edge: 45.0 °



The Lift force is: 0.0733379237479665 N

```
The  Lift  coefficient  is:    0.6011305225243155 at 10° AOA
The  Lift  curve  slope  is:   0.060113052252431555 per degree
The  Lift  curve  slope  is:   3.444224187713715 per radian

***************************************************************************
```



——————————————————————————————-

Problem 7.9 (part-a)

Quarter chord Sweep angle: 45°

Taper ratio: 1

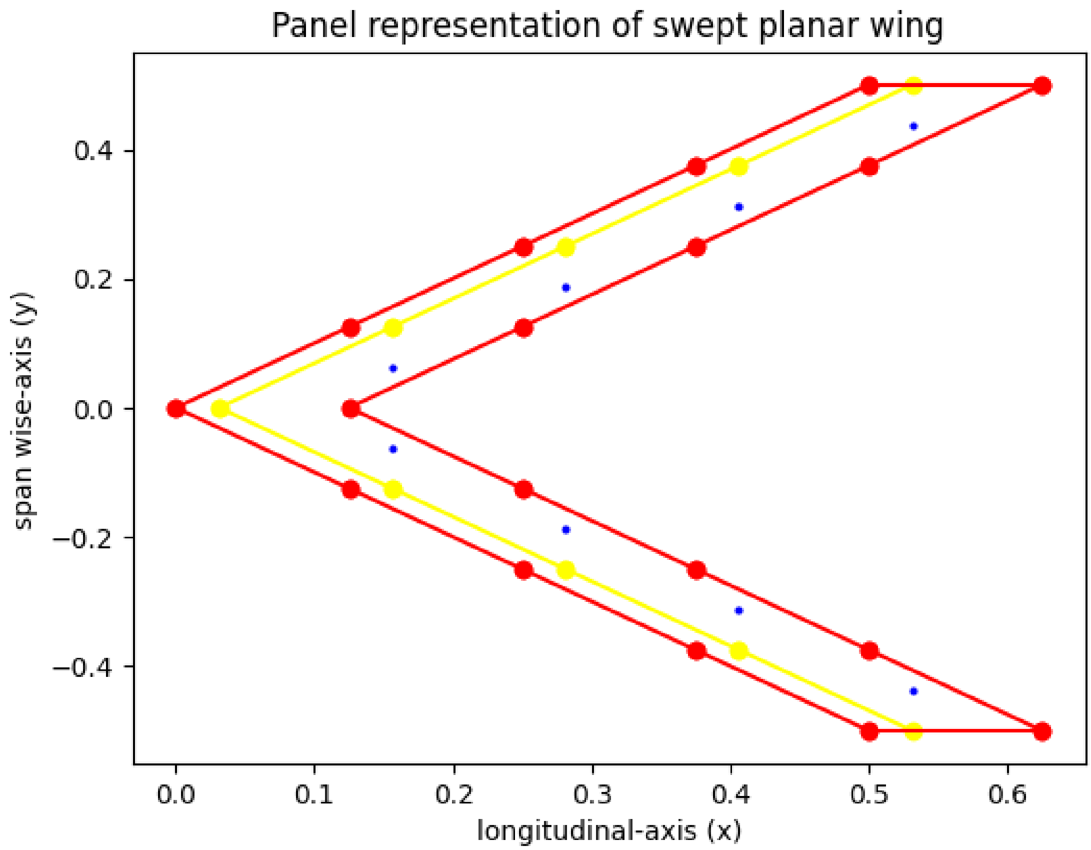
Aspect ratio: 8

No. of divisions along chord (nLtcx): 1

No. of divisions along span (nLtcy): 4

[473]:
```
# VLM(SwpCQrtr, TprRatio, AspctRatio, NLtcX, NLtcY)
VLM(-45, 1, 8, 1, 4)
```

```
The length of root chord:  0.125
The length of tip chord:  0.125
The chord length distribution from root to tip:  [0.125 0.125 0.125 0.125 0.125]
The sweep angle of wing leading edge:  45.0 °
```




```
The Lift force is:   0.05040227145814218  N
The Lift coefficient is:   0.6610133961723565 at 10° AOA
The Lift curve slope is:   0.06610133961723566 per degree
The Lift curve slope is:   3.7873277802285075 per radian

*******************************************************************
```

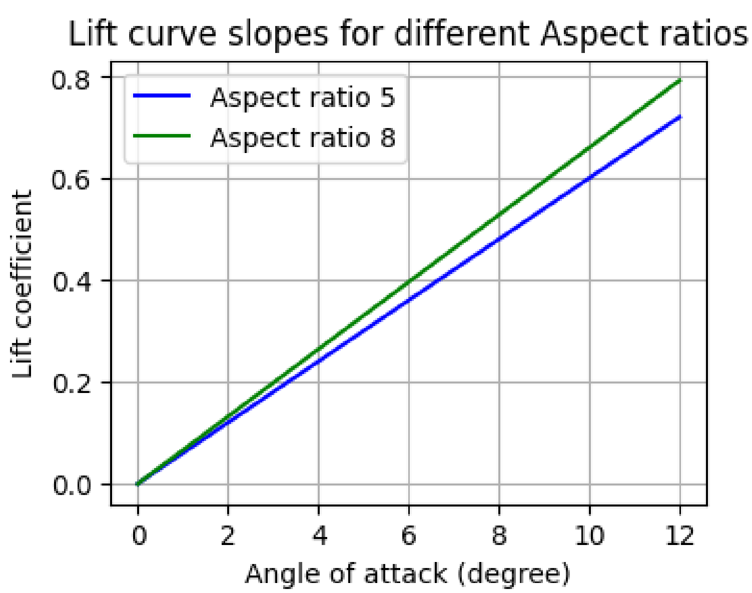
———————————————————————————-

Problem 7.9 (part-b)

Comparing lift curve slope of two aspect ratios from example 7.12 (AR=5) and problem 7.9 (AR=8)

[474]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Given slopes
slope1 = 3.444224187713715  #per radians
slope1 = slope1*np.pi/180    #per degree
slope2 = 3.7873277802285075 #per radians
slope2 = slope2*np.pi/180    #per degree
```

```
AR1=5
AR2=8

x = np.linspace(0, 12, 400)


# Assuming the wing section (airfoil) to be symmetric
y1 = slope1 * x
y2 = slope2 * x


plt.figure(figsize=(4, 3))
plt.plot(x, y1, color="blue", label=f"Aspect ratio {AR1}")
plt.plot(x, y2, color="green", label=f"Aspect ratio {AR2}")

plt.xlabel("Angle of attack (degree)")
plt.ylabel("Lift coefficient")
plt.title("Lift curve slopes for different Aspect ratios")

plt.legend()

plt.grid(True)
plt.show()
```



From the plot above it is evident that, as the Aspect ratio of the wing increases, the lift curve slope also increases.

——————————————————————————-

Problem 7.10

Quarter chord Sweep angle: 45°

Taper ratio: 0.5

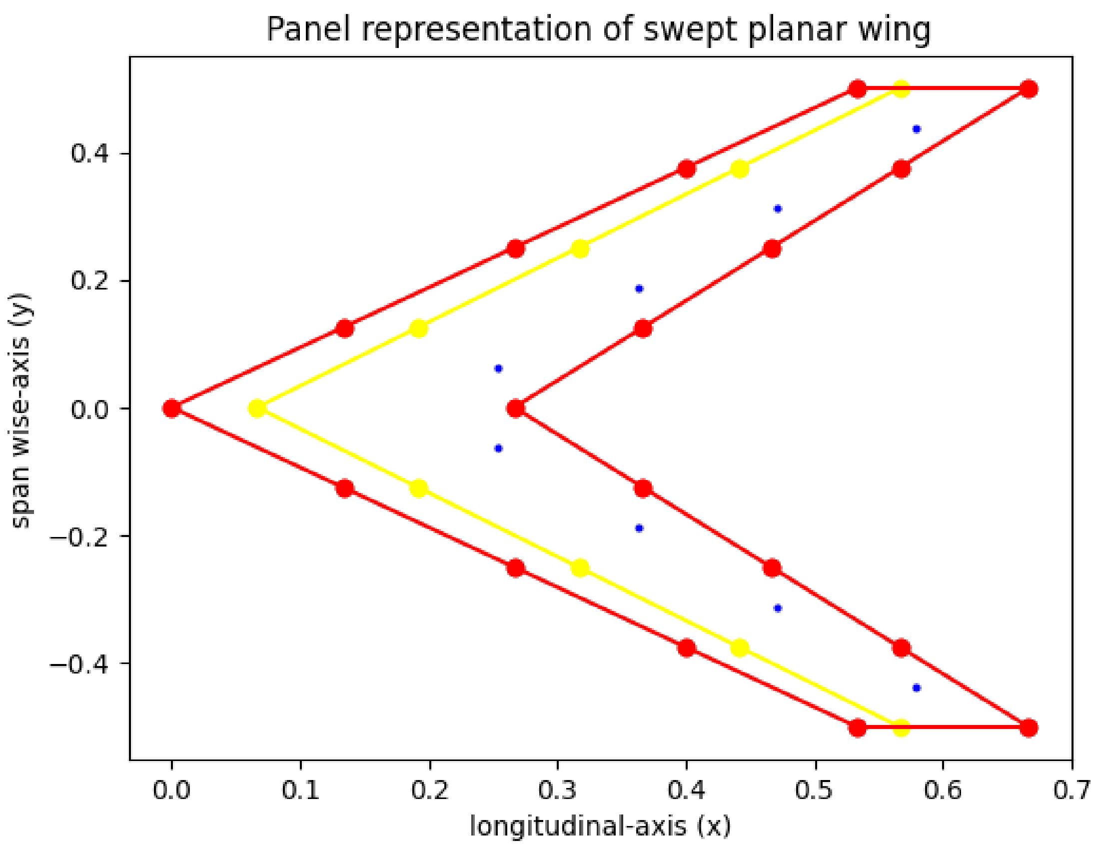
Aspect ratio: 5

No. of divisions along chord (nLtcx): 1

No. of divisions along span (nLtcy): 4

[475]:
```
# VLM(SwpCQrtr, TprRatio, AspctRatio, NLtcX, NLtcY)
VLM(-45, 0.5, 5, 1, 4)
```

The length of root chord: 0.26666666666666666
The length of tip chord: 0.13333333333333333
The chord length distribution from root to tip: [0.26666667 0.23333333 0.2
0.16666667 0.13333333]
The sweep angle of wing leading edge: 46.8476102659946 °



The Lift force is: 0.07616168644681202 N
The Lift coefficient is: 0.624276118416492 at 10° AOA

```
The Lift curve slope is:  0.0624276118416492 per degree
The Lift curve slope is:  3.5768386836074195 per radian

*****************************************************************
```

——————————————————————————-

Problem 7.11 (a)

Quarter chord Sweep angle: -45°

Taper ratio: 0.5

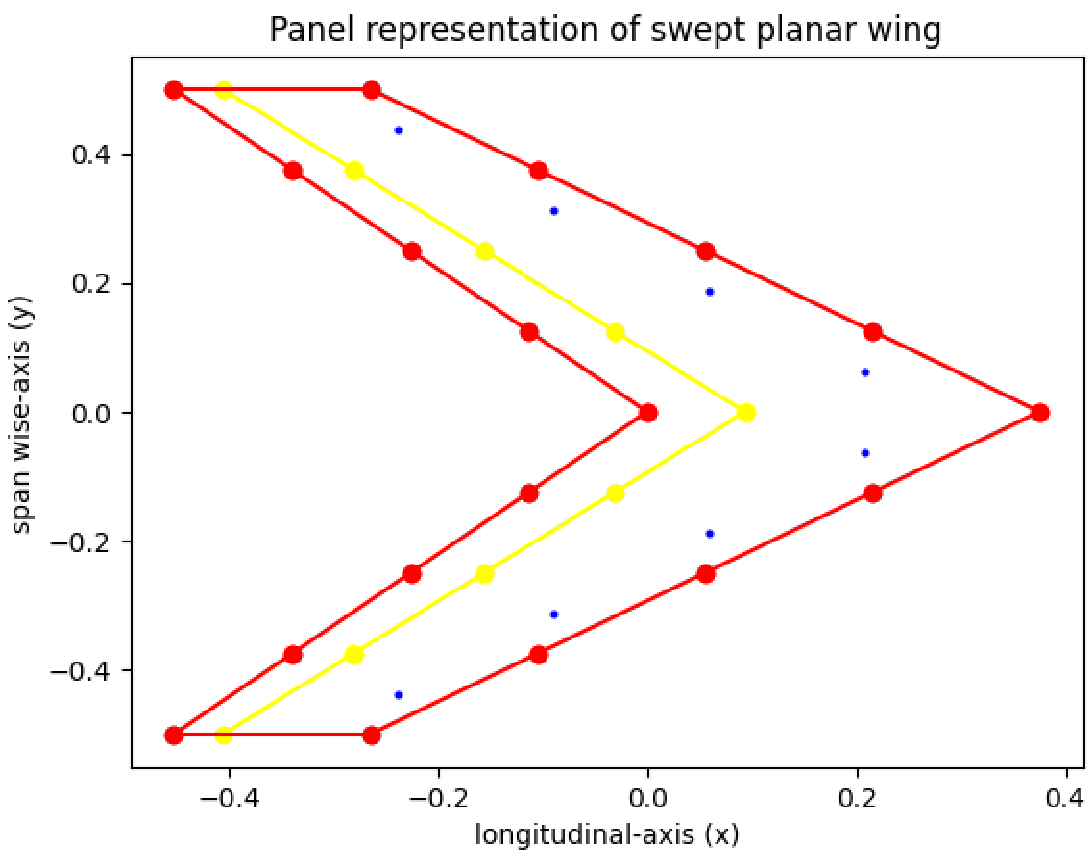
Aspect ratio: 3.55

No. of divisions along chord (nLtcx): 1

No. of divisions along span (nLtcy): 4

[476]:
```python
# VLM(SwpCQrtr, TprRatio, AspctRatio, NLtcX, NLtcY)
VLM(45, 0.5, 3.55, 1, 4)
```

```
The length of root chord:  0.37558685446009393
The length of tip chord:  0.18779342723004697
The chord length distribution from root to tip:  [0.37558685 0.3286385
0.28169014 0.23474178 0.18779343]
The sweep angle of wing leading edge:  -42.17982752755151 °
```



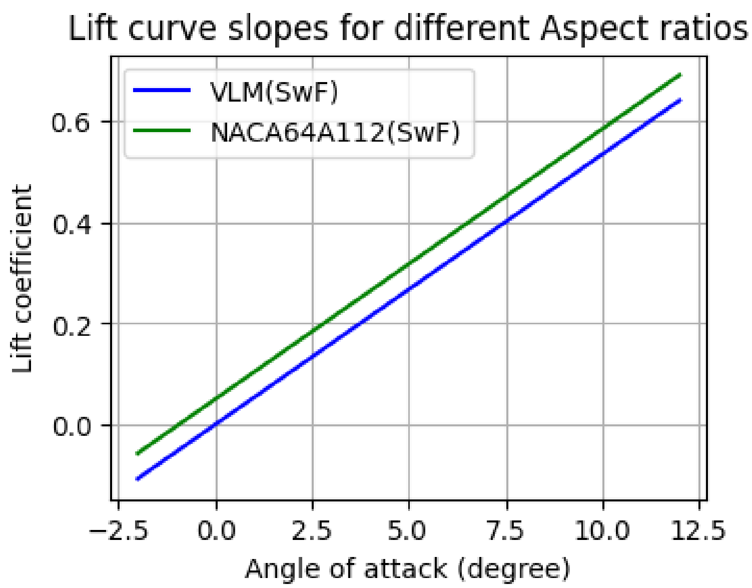

```
The Lift force is:  0.09178265200297178  N
The Lift coefficient is:  0.5341449419845079 at 10° AOA
The Lift curve slope is:  0.05341449419845079 per degree
The Lift curve slope is:  3.0604250823972516 per radian
```

****

Problem 7.11(b)

[477]:
```python
# Given slopes
slope1 = 3.0604250823972516  #per radians
slope1 = slope1*np.pi/180    #per degree

x = np.linspace(-2, 12, 400)


# Assuming the wing section (airfoil) to be symmetric
y1 = slope1 * x
y2 = slope1 * (x + 0.94)
```

```
plt.figure(figsize=(4, 3))
plt.plot(x, y1, color='blue', label= "VLM(SwF)")
plt.plot(x, y2, color='green', label="NACA64A112(SwF)")

plt.xlabel('Angle of attack (degree)')
plt.ylabel('Lift coefficient')
plt.title('Lift curve slopes for different Aspect ratios')

plt.legend()

plt.grid(True)
plt.show()
```



——————————————————————————-

Quarter chord Sweep angle: 45°

Taper ratio: 0

Aspect ratio: 1.5

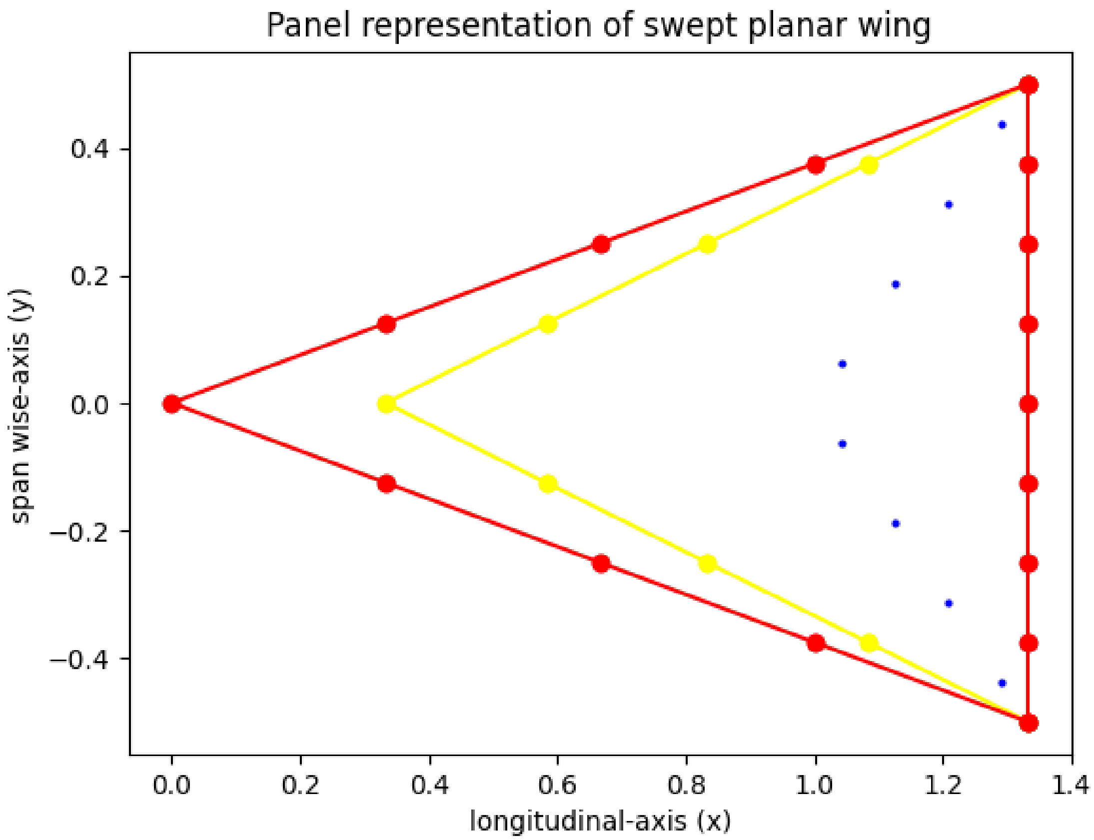
No. of divisions along chord (nLtcx): 1

No. of divisions along span (nLtcy): 4

[478]:
```
# VLM(SwpCQrtr, TprRatio, AspctRatio, NLtcX, NLtcY)
VLM(-45, 0, 1.5, 1, 4)
```

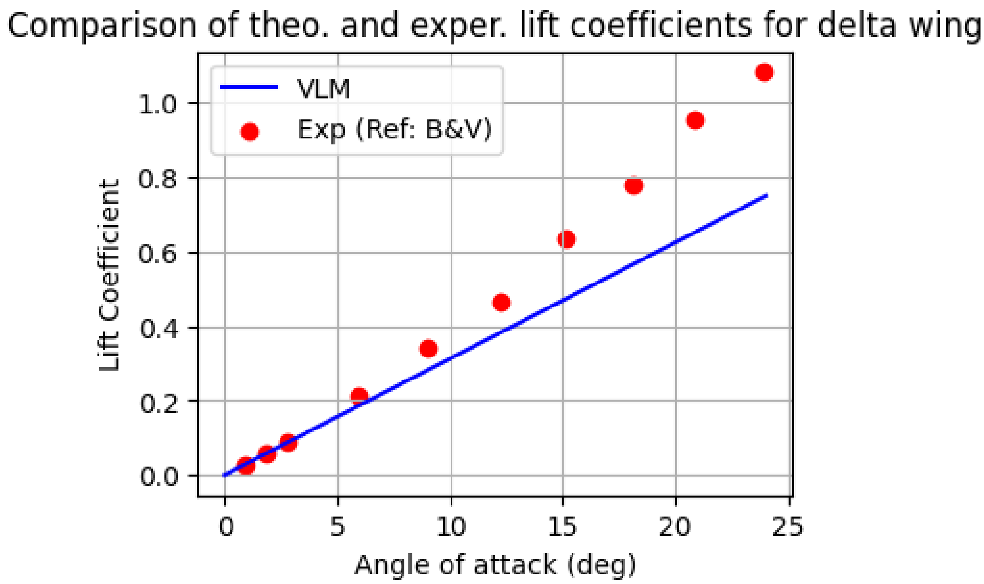
```python
# Comparing results with experimental values
Slope = 1.7907257007033237      # in per radians
Slope1 = Slope*np.pi/180       # in per degrees

x1 = np.linspace (0,24,400)
x2 = [0.9090, 1.8182, 2.7778, 5.9091, 8.9899, 12.2727, 15.1515, 18.0808,20.
↪8585, 23.9393]

# assuming the wing section as symmetric
y1 = Slope1*x1
y2 = [0.0294, 0.0591, 0.0888, 0.2136, 0.3424, 0.4673, 0.6337, 0.7823, 0.9546, 1.
↪0834]


plt.figure(figsize=(4,3))
plt.plot(x1,y1, color='blue', label = "VLM")
plt.scatter(x2,y2, color='red',marker='o', label = "Exp (Ref: B&V)")

plt.xlabel('Angle of attack (deg)')
plt.ylabel('Lift Coefficient')
plt.title('Comparison of theo. and exper. lift coefficients for delta wing')
plt.legend()
plt.grid(True)
plt.show()
```

```
The length of root chord:  1.3333333333333333
The length of tip chord:  0.0
The chord length distribution from root to tip:  [1.33333333 1.
0.66666667 0.33333333 0.        ]
The sweep angle of wing leading edge:  69.44395478041653 °
```



The Lift force is: 0.12709984187457915 N
The Lift coefficient is: 0.31254059477355534 at 10° AOA
The Lift curve slope is: 0.031254059477355553 per degree
The Lift curve slope is: 1.7907257007033237 per radian

****


Comparison of theo. and exper. lift coefficients for delta wing
VLM
Exp (Ref: B&V)
Lift Coefficient
Angle of attack (deg)
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25

Thin Airfoil Theory

```
import numpy as np
import warnings
import matplotlib.pyplot as plt
# defining a class

class airfoil(object):
# defining the constructor for the class named
  def___init__(self, airfoil, nsrs, nprs,aoa):
    # defining the variables aoan, prs, nsrs and airfoil as "self" to access these values in various functions
    theta=0.0
    self.nprs = nprs
    self.aoa = aoa
    self.nsrs = nsrs
    # dividing the chord line from theta =0 to pi into nprs equal points
    self.dtheta=np.pi/self.nprs
    # input string NACA airfoil series
    str_a=str(airfoil)
    # if the NACA airfoil is symmetrical
    if (int(str_a[0]) == 0 & int(str_a[1]) == 0):
      # this function will execute one set of functions
      self.execute1()
    else:
      # for NACA cambered airfoil
      if(len(str_a)==4):
        # finding max camber in 100ths of chord
        self.m=int(str_a[0])/100
        # finding position of max camber in 10ths of chord
        self.p=int(str_a[1])/10
        # conversion of max camber location from x coordinates to theta
        self.thetap=np.arccos(1-2*self.p)
      # this function will execute one set of functions
      self.execute()
```

```
  def fourier(self):
    A=0
    # Free stream velocity, V
    V=1
    # list to store all fourier coefficients
    self.An=[]
    # initializing theta to zero
    theta = 0
    # running the loop for all fourier components
    for i in range(self.nsrs):
      # calculating first fourier component for each theta varying from 0 to pi
      while(theta<=np.pi):
        # since the function dzc by dx is a piecewise function and changes at thetap (max camber location)
        if(theta<=self.thetap):
          # dzc by dx for theta less than thetap
          dzc_dx=self.m*(2*self.p-1+np.cos(theta))/self.p**2
        else:
          # dzc by dx for theta greater than thetap
          dzc_dx=self.m*(2*self.p-1+np.cos(theta))/(1-self.p)**2
        # this value dA is common for all fourier series
        dA=dzc_dx*np.cos(i*theta)*self.dtheta
        # for A0
        if(i==0):
          dAn=dA*(-1/np.pi)
```

```
        # for An
        else:
          dAn=dA*2/np.pi
        # adding up all fourier components values at each theta
        A=A+dAn
        theta=theta+self.dtheta
      # appending the fourier components A0, A1...An
      self.An.append(A)
      theta=0
      A=0
      print("A_",i," value is",self.An[i])
```

```
# to calculate the circulation and cp
  def circulationandcp(self):
  # printing circulation density at discretized points
    # arrays to store all CP values around upper and lower surface
    self.cp_u = []
    self.cp_l = []
    # to store all theta values from zero to pi incremented by dtheta
    self.theta2=[]
    # to store circulation at each theta value
    self.gamma_theta = []
    # initializing theta to zero to avoid garbage value
    theta=0
    # initializing free stream velocity
    V = 1
    while theta<=(np.pi-self.dtheta):
      # to calculate circulation for A0
      gammac = 2*V*self.An[0]*(1+np.cos(theta))/np.sin(theta)
      for i in range(self.nsrs-1):
        # to calculate circulation for rest of fourier series
        gammac = gammac + 2*V*self.An[i+1]*np.sin((i+1)*theta)
      # contribution of freestream at some angle of attack to the circulation
      gammac = gammac + 2*V*self.aoa*np.pi/180*(1+np.cos(theta))/np.sin(theta)
      # calculating the coeff of pressure at upper and lower surface using circulation and
velocity
      cp_u0=-gammac/V
      cp_l0=gammac/V
      warnings.filterwarnings("ignore",category=RuntimeWarning)
      # storing circulation, discretized theta values, coeff of pressure at upper and lower
surface values at each theta
      self.gamma_theta.append(gammac)
      self.cp_u.append(cp_u0)
      self.cp_l.append(cp_l0)
      self.theta2.append(theta)
      theta = theta + self.dtheta
# calculating lift coefficient and moment coefficient at quater chord point
```

```
    cl=2*np.pi*self.aoa*np.pi/180+np.pi*(2*self.An[0] + self.An[1])
    cmc_4 = -np.pi/4*(self.An[1] - self.An[2])
```

```
    # Printing lift coeff and moment coeff about c/4
    print("cl from Camber problem is ", cl)
    print("cm_c/4 from Camber problem is ", cmc_4)
```

```
# function to store each theta value incremented by discretized theta
  def thetaarray(self):
    self.theta1=[]
    theta=0
    self.x5=[]
```

```
    while(theta<=(np.pi-self.dtheta)):
      theta=theta+self.dtheta
      dxx=0.5*(1-np.cos(theta))
      self.theta1.append(theta)
      # also storing the theta values along chord in terms of x in list x5
      self.x5.append(dxx)
```

```
# printing camber function without fourier components
  def theoreticalcamber(self):
    # creating a list to store all camber function values ZC
    self.zc2=[]
    # since the zc is a piece wise function and changes after theta = thetap, i.e at max
camber location
    for x in range(self.nprs):
      if(self.x5[x]<=self.p):
        zc=(self.m*(2*self.p*self.x5[x]-self.x5[x]**2)/self.p**2)
        self.zc2.append(zc)
      elif(self.x5[x]>self.p):
        zc = (self.m*(1-2*self.p+2*self.p*self.x5[x]-self.x5[x]**2)/(1-self.p)**2)
        self.zc2.append(zc)
# appending each camber function value into list ZC2
```

```
# printing camber function with fourier components
  def practicalcamber(self):
    self.zc_=[]
    z=0
    zc1=0
    theta=0
    self.cmbr=[]
    sum=0
    # fig, ax = plt.subplots()
    for i in range(self.nsrs):
      zc0=[]
      for j in range(len(self.theta1)):
        z=0
        theta=0
        while(theta<=self.theta1[j]):
          if i==0:
            zc1 =zc1 + (-1)*self.An[i]*0.5*np.sin(theta)*self.dtheta
          else:
            zc1 =zc1 + self.An[i]*0.5*np.cos(i*theta)*np.sin(theta)*self.dtheta
          theta=theta+self.dtheta
        zc0.append(zc1)
        zc1=0
      self.zc_.append(zc0)        #saving a 2D array having values of fourier components A0,
A1,... at each theta from zero to pi
      plt.plot(self.theta1,zc0)    #to plot fourier components curves

    for i in range(self.nprs):
      for j in range(self.nsrs):
        sum=sum+self.zc_[j][i]    #appending the values into list camber having zc values
obtained from fourier coeffs
      self.cmbr.append(sum)
      sum=0
```

```
# calculating coefficient of pressure and circulation density for symmetric airfoil
  def symmetric(self):
```

```
 # printing circulation density at discretized points
   self.cp_u = []
   self.cp_l = []
   self.theta2=[]
   self.gamma_theta = []
   theta=self.dtheta
   V = 1

   while theta<=(np.pi-self.dtheta):
     gammac = 2*V*self.aoa*np.pi/180*(1+np.cos(theta))/np.sin(theta)
     cp_u0=-gammac/V
     cp_l0=gammac/V
     self.gamma_theta.append(gammac)
     self.cp_u.append(cp_u0)
     self.cp_l.append(cp_l0)
     self.theta2.append(theta/np.pi)
     theta = theta + self.dtheta
   cl = 2*np.pi*self.aoa*np.pi/180
   cmc_4 = 0
    # Printing lift coeff and moment coeff about c/4 and circulation at discretized points of
theta
   print("cl from Camber problem is ", cl)
   print("cm_c/4 from Camber problem is ", cmc_4)
   print("circulation", self.gamma_theta)
```

```
# function to plot theoretical zc curve with x variation and zc in terms of fourier components
  def plot(self):
```

```
   plt.figure(figsize=(12,2))  # Optional: Set the figure size
   plt.plot(self.x5, self.zc2, label='theoretical', color='blue', linestyle='--')
   plt.plot(self.x5, self.cmbr, label='fourier components', color='green', linestyle='-')
   # Plot the curve
   plt.title('camber plot') # Set the title of the plot
   plt.xlabel('x/c') # Label for the x-axis
   plt.ylabel('zc')  # Label for the y-axis
   plt.grid(True)  # Display a grid
   plt.legend()  # Display a legend
```

```
   fig, ax = plt.subplots()
   # Create the plot
   plt.gca().invert_yaxis()
   plt.plot(self.theta1 , self.cp_l , color='blue')
   plt.plot(self.theta1 , self.cp_u ,color='red')
   plt.legend()
   # Plot the data and specify labels
   ax.plot(self.theta1, self.cp_l, label='Cp_lower')
   ax.plot(self.theta1, self.cp_u, label='Cp_upper')
   # Add a legend to the plot
   ax.legend()
   # plt.show()
```

```
  def plot1(self):
```

```
    fig, ax = plt.subplots()
    # Create the plot
    plt.gca().invert_yaxis()
    plt.plot(self.theta1 , self.cp_l , color='blue')
    plt.plot(self.theta1 , self.cp_u ,color='red')
```

```
    # Plot the data and specify labels
    ax.plot(self.theta1, self.cp_l, label='Cp_lower')
    ax.plot(self.theta1, self.cp_u, label='Cp_upper')
    ax.legend()
    plt.show()
```

```
# Add a legend to the plot

  def execute(self):
```

```
    self.fourier()
    self.thetaarray()
    self.circulationandcp()
    self.theoreticalcamber()
    self.practicalcamber()
    self.plot()
```

```
  def execute1(self):
```

```
    self.thetaarray()
    self.symmetric()
    self.plot1()
# inputing the data from user for NACA type of airfoil, no. of discretized points, number of fourier series coefficients and angle of attack
print("Enter the NACA airfoil series: ")
str_a=input()
print("Enter the number of Fourier series components: ")
nsrs=int(input())
print("Enter the number of discretized points: ")
nprs=int(input())
print("Enter the angle of attack: ")
aoa=int(input())
# to call the constructor of the class
airfoil(str_a,nsrs+1,nprs,aoa)
```

```
# a = airfoil("2412",10,100,5)

#Here we are defining the class to plot the streamline of particular case using the circulation density solution
```

```
class streamline(object):
  def__init__(self,b):
    #Taking input of extreme coordinates to create the border/limit of the streamline plot grid.
    print("Enter the left bottom most x co-ordinate of grid")
    x0=int(input())
    print("Enter the left bottom most y co-ordinate of grid")
    z0=int(input())
    print("Enter the right top most x co-ordinate of grid")
    x_=int(input())
    print("Enter the right top most y co-ordinate of grid")
    z_=int(input())
```

```
    ds=0.01 #the size of the discretized grid element for numerical integration along the 
streamline.
    zy=[] #zy is zeta which is arbitrary variable along x.
    v=1 #v is free stream velocity in m/s. (considered v=1m/s here)
    self.b  = a #Assigning airfoil value (which is "a") to the object "b".

    gammax=[] #Creating the empty array which is made for circular density of the vortex sheet.

  #Creating for loop to find the total circulation density of vortex sheets.
    for i in range(len(self.b.x5)):
      gamma1=2*self.b.aoa*np.pi/180*v*np.sqrt((self.b.x5[len(self.b.x5)-1]-
self.b.x5[i])/self.b.x5[i])
      gammax.append(gamma1)
      zy.append(self.b.x5[i])

    x=x0 #initalizing "x=x0"
    z=z0 #initalizing "z=z0"
    u1=0 #initalizing instantaneous tangential velocity component "u1" and equationg to zero.
    w1=0 #initalizing instantaneous perpendicular velocity component "u1" and equationg to 
zero.
    i=0 #initial iteration value "i" is equated to zero
    x1=[] #creating empty array to access the instantaneous "x" coordinate.
    z1=[] #creating empty array to access the instantaneous "z" coordinate.
    zf =(z_-z)/90
    z2     =     z0
    while(z <= z_):
      while(x<=x_):
        while(zy[i]<=zy[len(zy)-2]):
          if(zy[i]==zy[len(zy)-1]):
            dzy=0.001         #if i reaches the last value , increment value (i+1) will result 
in error therefore assigning dzy to a random value 0.001
          else:
            dzy=zy[i+1]-zy[i]    # dzy is the difference between the incremented values of z
          du=gammax[i]/(2*np.pi)*(z*dzy/((x-zy[i])**2+z**2))             #calculating x 
component velocity contribution from vortex
          dw=-1*gammax[i]/(2*np.pi)*(x-zy[i])*dzy/((x-zy[i])**2+z**2)     #calculating z 
component velocity contribution from vortex
          u1=u1+du
          w1=w1+dw
          i=i+1
        U=u1+v*np.cos(self.b.aoa*np.pi/180)  #contribution of x component of free stream 
velocity to the velocity due to vortex
        W=w1+v*np.sin(self.b.aoa*np.pi/180) #contribution of z component of free stream 
velocity to the velocity due to vortex
        thetax=np.arctan(W/U)   #calculating the theta values
        x=x+ds*np.cos(thetax)   #calculating next x coordinate using theta
        z=z+ds*np.sin(thetax)  ##calculating next z coordinate using theta
        # initializing the velocities to zero to calculate new velocities at next location
        w1=0
        u1=0
        i=0
        #storing the coordinates of stream function psi in x1 and z1
        x1.append(x)
        z1.append(z)
```

```
      plt.plot(x1,z1, color = 'black')
      z = z2 + zf
      z2 = z2 + zf
      x1=[]
      z1=[]
      x=x0
```

```
b = streamline(a)
```

OUTPUT:
Enter the NACA airfoil series: 4412
Enter the number of Fourier series components: 10
Enter the number of discretized points: 100
Enter the angle of attack: 2
A_ 0 value is -0.009320711078221555
A_ 1 value is 0.16632427785174397
A_ 2 value is 0.02838628086848422
A_ 3 value is 0.00887597072084825
A_ 4 value is -0.0035449913504861374
A_ 5 value is 0.0003642467094368959
A_ 6 value is 0.0016224591934828987
A_ 7 value is 0.0051031060934518535
A_ 8 value is 0.0006864032389936017
A_ 9 value is 0.002279050507897977
A_ 10 value is 0.00030943879071538465
cl from Camber problem is 0.6832839167599489
cm_c/4 from Camber problem is -0.10833624949337498