```
import math
import scipy.integrate as it
import matplotlib.pyplot as plt
import numpy as np

"""
Defining constants such as length

"""
L = 1290+1 #add one to lengh to make it possible to iterate to the 1290 index of an array
E = 4000
MU = 0.2

"""
initialy define sfd as a 2-D array of length L
an sfd array entry will be in the following form
[values of the shear force, type of point IE pointload, support, nothing]
in the array value 0 corresponds to no support or pointload at that point
            value 1 corresponds to a support
            value 2 corresponds to a pointload
"""
sfd = [[0.0, 0.0]]*L
sfd = np.array(sfd)
support_x = 15
support_x_right = 1075
point_loads = []
bmd = np.array([0.0]*L)

"""
sigmas arrays are to collect the indivsual buckling force in the webs, side flange, and mid flang
"""
sigmas1 = []
sigmas2 = []
sigmas3 = []

def buildSFD(xP, P):
    """
    functions to construct SFD

        parameters::
            xP(int): the position of the point load
            P(int): value in N of the pointload

        functions adds inputed pointload to a global array of pointloads from there the reaction forces are calculated
        the function then checks each point of the sfd array and if it is at a pointload it adds the corresponding reaction force
            and if it at a pointload the correspnding value is substracted from the total
        returns:
            an array that includes only the value of the sheer force at each point in the bridge

    """
    global sfd
    global point_loads
    temp = [P, xP]
    added = False
    for i in range(len(point_loads)):
        if point_loads[i][1] == temp[1]:
            point_loads[i][0] = temp[0]
            added = True
    if not added:
        point_loads.append(temp)
    sum1  = 0
    sum2 = 0
    # points loads will be a 2d array with [P, xp] as values

    for i in range(len(point_loads)):
        sfd[point_loads[i][1]] = point_loads[i][0]
```

```
        sum1 += point_loads[i][0] * (point_loads[i][1] - support_x)
        sum2 += point_loads[i][0]
    # by = (xP-support_x) * P / (support_x_right - support_x)
    by = sum1/(support_x_right-support_x)
    ay = sum2 - by

    sfd[support_x] = [ay, 1]
    sfd[support_x_right] = [by, 1]

    sum = 0.0
    for i in range(0, L):
        # print(sfd[i])
        # print(sfd[i][0])
        if (sfd[i][1] == 0):
            pass
        elif (sfd[i][1] ==1):
            sum += sfd[i][0]
        else:
            sum-= sfd[i][0]
        sfd[i][0] = sum

    arr = []
    for i in range(len(sfd)):
        arr.append(sfd[i][0])
    return arr



def printSFD():
    """
    functions to display a matplot.lib graph of the SFD
    """
    plt.title("Bridge SFD for P=617N, Pfail = 617N")
    plt.xlabel("x (mm)")
    plt.ylabel("shear force (N)")
    arr = []
    for i in range(len(sfd)):
        arr.append(sfd[i][0])
    zeroliney = [0, 0]
    zerolinex = [0, L]
    plt.plot(zerolinex, zeroliney)
    plt.plot(arr)
    plt.show()


def buildBMD():
    """
    functions that builds the BMD using the sheer force values from the array SFD
    the BMD is then calculated by using a function from the sciPy library cumtrapz
    this function calculated the integral of a list using cumulative trapezoidal sums
    """
    global bmd
    arr = []
    for i in range(len(sfd)):
        arr.append(sfd[i][0])

    bmd=it.cumtrapz(arr)


def printBMD():
    """
    functions to display BMD using Matplot.lib
    """
    plt.title("Bridge BMD from P=617, PFail = 617")
    plt.xlabel("x (mm)")
    plt.ylabel("bending moment (Nmm)")
```

```
    zeroliney = [0, 0]
    zerolinex = [0, L]
    plt.plot(zerolinex, zeroliney)

    plt.gca().invert_yaxis()
    plt.plot(bmd)
    plt.show()



def ybar(height, widthtop, widthbottom, topthickness, bottomthickness, rightthickness, leftthickness, shape, tabThickness):
    """
    function to calculate the yBar of a crosssection

        parameters:
            height(int): height of the cross section
            widthtop(int): width of the top member
            widthbottom(int): width of bottom of the crosssection
            topthickness(int): thickenss of top memebr
            bottomthickness(int): thickness of bottom member
            rightthickness(int): thickness of right memeber
            leftthickness(int): thickness of left memebr
            shape(int): which cross section is being used IE for different cross sections a different number is given
            tabThicknes(int): thickness of the tabs
        returns:
            ybar(float): distance to the centroid
    """
    if(shape == 0):#shape = 0 during test bridge
            area1 = widthtop * topthickness #top part of bridge
            area2 = 10 * (tabThickness) #tabs 10 is tab width
            area3 = leftthickness * (height - topthickness)
            area4 = (widthbottom - 2*leftthickness) * bottomthickness
            d1 = (height-topthickness/2)
            d2 = (height - topthickness - tabThickness/2)
            d3 = (height-topthickness)/2
            d4 = bottomthickness/2
            y_bar = area1*d1 + 2*(area2*d2) + 2*(area3*d3) + area4*(d4)
            y_bar = y_bar / (area1+2*area2+2*area3+area4)
            return y_bar

def I0(b, h):
    """
    function to calculate the inital second moment of area
        paramaeters
            b(float): wifth
            h(float): height
        returns I0(double): second moment of area
    """
    I0 = (b * h * h * h)/12
    return I0

def second_moment_of_area(height, widthtop, widthbottom, topthickness, bottomthickness, rightthickness, leftthickness, shape,
tabThickness):
    """
    function to calculate the Second moment of area of a crosssection

        parameters:
            height(int): height of the cross section
            widthtop(int): width of the top member
            widthbottom(int): width of bottom of the crosssection
            topthickness(int): thickenss of top memebr
            bottomthickness(int): thickness of bottom member
            rightthickness(int): thickness of right memeber
            leftthickness(int): thickness of left memebr
            shape(int): which cross section is being used IE for different cross sections a different number is given
            tabThicknes(int): thickness of the tabs
        returns:
```

```
        second_moment_of_are(float): second_moment_of_area
    """
    centroid = ybar(height, widthtop, widthbottom, topthickness, bottomthickness, rightthickness, leftthickness, shape,
tabThickness)
    if shape == 0:
        area1 = widthtop * topthickness #top part of bridge
        area2 = 10 * (tabThickness) #tabs 10 is tab width
        area3 = leftthickness * (height - topthickness)
        area4 = (widthbottom - 2*leftthickness) * bottomthickness
        d1L = (height-topthickness/2)
        d2L = (height - topthickness - tabThickness/2)
        d3L = (height-topthickness)/2
        d4L = bottomthickness/2
        d1C = d1L - centroid
        d2C = d2L - centroid
        d3C = d3L - centroid
        d4C = d4L - centroid

        term1 = I0(widthtop, topthickness) + (area1 * (d1C)**2) #top
        term2 = I0(10, tabThickness) + area2 * (d2C)**2 #tabs
        term3 = I0(leftthickness, height - topthickness) + (area3 * (d3C)**2)
        term4 = I0(widthbottom - 2*leftthickness, bottomthickness) + (area4 * (d4C)**2)
        second_moment_of_area = term1 + 2* term2 + 2*term3 + term4


    return second_moment_of_area



def first_moment_of_area(height, widthtop, widthbottom, topthickness, bottomthickness, rightthickness, leftthickness, shape,
tabThickness):
    """
    function to calculate the first moment of area of a crosssection

        parameters:
            height(int): height of the cross section
            widthtop(int): width of the top member
            widthbottom(int): width of bottom of the crosssection
            topthickness(int): thickenss of top memebr
            bottomthickness(int): thickness of bottom member
            rightthickness(int): thickness of right memeber
            leftthickness(int): thickness of left memebr
            shape(int): which cross section is being used IE for different cross sections a different number is given
            tabThicknes(int): thickness of the tabs
        returns:
            first_moment_of_area(float): first moment of area
    """
    centroid = ybar(height, widthtop, widthbottom, topthickness, bottomthickness, rightthickness, leftthickness, shape,
tabThickness)

    if shape == 0:
        area1 = widthbottom*bottomthickness
        area2 = (centroid - bottomthickness) * rightthickness
        area3 = (centroid - bottomthickness) * leftthickness
        d1 = centroid - (bottomthickness/2)
        d2 = (centroid - bottomthickness)/2
        d3 = (centroid - bottomthickness)/2

        Q = area1*d1 + area2*d2 + area3*d3

    return Q

def get_properties():
    """
    function to set the sectional properties of the bridge
    the properties are stored in arrays of length L asto be able to set the various properties at each point in the bridge
```

```
        returns:
            height(int): height of the cross section
            widthtop(int): width of the top member
            widthbottom(int): width of bottom of the crosssection
            topthickness(int): thickenss of top memebr
            bottomthickness(int): thickness of bottom member
            rightthickness(int): thickness of right memeber
            leftthickness(int): thickness of left memebr
            shape(int): which cross section is being used IE for different cross sections a different number is given
            tabThicknes(int): thickness of the tabs
    """
    height = [75+35]*L
    widthTop = [100]*L
    widthBottom = [80]*L
    topThickness = [1.27*2]*L


    bottomThickness = np.concatenate(([1.27]*520, [1.27*2]*50, [1.27]*570, [1.27*2]*(L-520-50-570)), axis = 0) #double bottom
thickness between 520 - 570 and 1140 - the end
    rightThickness = np.concatenate(([1.27]*540, [1.27 * 2]*10, [1.27]*(L-540-10)), axis = 0) #double thickness between 540-550
    leftThickness = np.concatenate(([1.27]*540, [1.27 * 2]*10, [1.27]*(L-540-10)), axis = 0) #double thickness between 540-550

    shape = [0] * L #similar crosssection through the entire length
    tabThickness = [1.27]*L
    return  height, widthTop, widthBottom, topThickness, bottomThickness, rightThickness, leftThickness, shape, tabThickness

def y_top(y_bar, height):
    """
    calculates y_top
        paramarers:
            y_bar(float): centroid location
            height(int): height of cross section
        returns:
            y_top(float): distance from centroid to top of crosssection
    """
    return y_bar - height

def geometric_properties():
    """
    functions to generate lists of geometric properties
    each properties is a list, length L with each entry being the value of that property at each point on the bridge

    returns
        I(list): second moment of area at each point on the bridge
        y_bar(list): centroid locartion at each location on the bridge
        Q(list): first moment of area at each location on the bridge
        yTop(list): distance from centroid to top of crosssection at each point on the bridge
    """
    height, widthtop, widthbottom, topthickness, bottomthickness, rightthickness, leftthickness, shape, tabThickness=
get_properties()
    y_bar = []
    I = []
    Q = []
    yTop = []

    for i in range(L):
        y_bar.append(ybar(height[i], widthtop[i], widthbottom[i], topthickness[i], bottomthickness[i], rightthickness[i], leftthickness[i],
shape[i], tabThickness[i]))
        I.append(second_moment_of_area(height[i], widthtop[i], widthbottom[i], topthickness[i], bottomthickness[i], rightthickness[i],
leftthickness[i], shape[i], tabThickness[i]))
        Q.append(first_moment_of_area(height[i], widthtop[i], widthbottom[i], topthickness[i], bottomthickness[i], rightthickness[i],
leftthickness[i], shape[i],tabThickness[i]))
        yTop.append(height[i]-y_bar[i])


    return I, y_bar, Q, yTop
```

```
def V_fail(Tau):
    """
    functiont to calculate the values of sheer needed to cause shear failure
        parameters:
            Tau(int): shear strength
        returns:
            V_fail_values(list): the values of sheer that would cause a shear failure
    """

    I = geometric_properties()[0]
    Q = geometric_properties()[2]
    centroid = geometric_properties()[1]
    bottomthickness = get_properties()[4]
    height = get_properties()[0]
    topthickness = get_properties()[3]
    widthbottom = get_properties()[2]
    widthtop = get_properties()[1]
    rightthickness = get_properties()[5]
    leftthickness = get_properties()[6]
    V_fail_values = []

    for i in range(L):
        if 0 <= centroid[i] < bottomthickness[i]:
            centroidlocation = "bottom"
        elif bottomthickness[i] <= centroid[i] < height[i] - topthickness[i]:
            centroidlocation = "middle"
        elif height[i] - topthickness[i] <= centroid[i] < height[i]:
            centroidlocation = "top"

        if centroidlocation == "bottom":
            b = widthbottom[i]
        elif centroidlocation == "middle":
            b = rightthickness[i] + leftthickness[i]
        elif centroidlocation == "top":
            b = widthtop[i]

        V_fail = (I[i]*Tau*b)/Q[i]

        V_fail_values.append(V_fail)

    return V_fail_values

def V_buck(max_diaphragm_distance):
    """
    functiont to calculate the values of sheer needed to cause shear buckling
        parameters:
            max_diaphragm_distance(int): max distance between any two diaphgram
        returns:
            V_buck_values(list): the values of sheer that would cause a failure due to shear buckling
    """

    height = get_properties()[0]
    widthtop = get_properties()[1]
    widthbottom = get_properties()[2]
    topthickness = get_properties()[3]
    bottomthickness = get_properties()[4]
    rightthickness = get_properties()[5]
    leftthickness = get_properties()[6]
    I = geometric_properties()[0]
    Q = geometric_properties()[2]
    centroid = geometric_properties()[1]
    V_buck_values = []

    for i in range(L):

        Tcrit = ((5*((math.pi)**2)*E)/(12*(1-(MU**2)))) * (((rightthickness[i]/max_diaphragm_distance)**2) +
```

```
((leftthickness[i]/(height[i]- topthickness[i]))**2))

        if 0 <= centroid[i] < bottomthickness[i]:
            centroidlocation = "bottom"
        elif bottomthickness[i] <= centroid[i] < height[i] - topthickness[i]:
            centroidlocation = "middle"
        elif height[i] - topthickness[i] <= centroid[i] < height[i]:
            centroidlocation = "top"

        if centroidlocation == "bottom":
            b = widthbottom[i]
        elif centroidlocation == "middle":
            b = rightthickness[i] + leftthickness[i]
        elif centroidlocation == "top":
            b = widthtop[i]

        V_buck = Tcrit*I[i]*b/Q[i]

        V_buck_values.append(V_buck)

    return V_buck_values

def M_failMatT(sigT):

    """
    calculate  the moment that causes tension failure at each x
        returns:
            moment_fail(list): list of momement that would cause the bridge to break at each location along the bridge

    """
    # change with right indices

    I = geometric_properties()[0]
    y_bar = geometric_properties()[1]

    y_top = geometric_properties()[3]

    # ############################
    moment_fail = []
    for i in range(L-1):
        if bmd[i] > 0: #positive moment the max tension will be at the bottom
            moment_fail.append((sigT * I[i])/y_bar[i]) #M = sigma*I / y
        elif bmd[i] <0:
            moment_fail.append(-1 * sigT * I[i] /y_top[i])
        else:
            moment_fail.append(0)

    return moment_fail



def M_failMatC(sigC):
    global L
    """
    calculate the moment the causes compression failure at each x
        returns:
            moment_fail(list): array of the moment required to break bridge at each location

    """
    # change with right indices
    I = geometric_properties()[0]
    y_bar = geometric_properties()[1]
    Q = geometric_properties()[2]
    y_top = geometric_properties()[3]
    # ############################
```

```
    moment_fail = []
    for i in range(L-1):
        if bmd[i] >0: #positive moment the max tension will be at the bottom
            moment_fail.append(sigC * I[i]/y_bar[i]) #M = sigma*I / y
        elif bmd[i] <0:
            moment_fail.append(-1 * sigC * I[i] /y_top[i])
        else:
            moment_fail.append(0)
    return moment_fail

def MFailBuck(t):

    """
    calculated moment needed to break bridge due to buckling at every location along the bridge
    checks wich form of buckling is most prevelant:
        mid flange buckling
        side flange buckling
        web compression buckling
    it will then add the corresponding moment to list moments

    parameters:
        t(float): thickness
    returns:
        moments(list): list of moments required to cause buckling failure at each point on the bridge


    3 cases
        case 1 when two sides restained constant force k = 4
        case 2 when one side restained constant froce k =0.425
        case 3 carries compressive stresses (like the webs) k=6

        sigma = (4*pi**2*E)/(12(1-MU**2))*(t/b)**2
        M = sigma * I /Y
        b= width
    """


    global sigmas1, sigmas2, sigmas3

    I = geometric_properties()[0]
    y_bar = geometric_properties()[1]
    y_top = geometric_properties()[3]
    topthickness = get_properties()[3]
    sigma = 0
    moments = []
    shape = get_properties()[7]
    for i in range(0, L-1):
        if shape[i] == 0: #the demo bridge
            if(bmd[i] > 0):
                sigma1 = case1(i, topthickness[i], 80)
                n1 =  sigma1*I[i] / y_top[i]
                n2 = sigma1*I[i] / y_bar[i]
                sigmas1.append(min(n1,n2))
                # sigma1 = case1(i, t, 80)
                sigma2 = case2(i, t,10)

                n1 =  sigma2*I[i] / y_top[i]
                n2 = sigma2*I[i] / y_bar[i]
                sigmas2.append(min(n1,n2))

                sigma3 = case3(i, t, 32.02)
                n1 =  sigma3*I[i] / y_top[i]
                n2 = sigma3*I[i] / y_bar[i]
                sigmas3.append(min(n1,n2))
            elif bmd[i]<0:
                sigma1 = case1(topthickness[i], 80)
```

```
            n1 =  -1* sigma1*I[i] / y_top[i]
            n2 = -1* sigma1*I[i] / y_bar[i]
            sigmas1.append(min(n1,n2))
            # sigma1 = case1(i, t, 80)
            sigma2 = case2(t,10)

            n1 =  -1*sigma2*I[i] / y_top[i]
            n2 = -1*sigma2*I[i] / y_bar[i]
            sigmas2.append(min(n1,n2))

            sigma3 = case3(t, 32.02)
            n1 =  -1*sigma3*I[i] / y_top[i]
            n2 = -1*sigma3*I[i] / y_bar[i]
            sigmas3.append(min(n1,n2))
        else:
            sigma1 = case1(topthickness[i], 80)
            n1 =  -1* sigma1*I[i] / y_top[i]
            n2 = -1* sigma1*I[i] / y_bar[i]
            sigmas1.append(0)
            # sigma1 = case1(i, t, 80)
            sigma2 = case2(i, t,10)

            n1 =  -1*sigma2*I[i] / y_top[i]
            n2 = -1*sigma2*I[i] / y_bar[i]
            sigmas2.append(0)

            sigma3 = case3(t, 32.02)
            n1 =  -1*sigma3*I[i] / y_top[i]
            n2 = -1*sigma3*I[i] / y_bar[i]
            sigmas3.append(0)



        sigma = min(sigma1, sigma2, sigma3)
        # take min of potential sigmas calculated like if a memeber uses both case1 and 2
        n1 =  sigma*I[i] / y_top[i]
        n2 = sigma*I[i] / y_bar[i]
      moments.append(min(n1, n2))

  return(moments)

def case1(t, b):
  n1 = (4*math.pi**2 * E)
  n2 = (12*(1-MU**2))
  n3 = (t/b)**2
  return n1/n2 * n3

def case2(t, b):

  sigma = (0.425*math.pi**2 * E)/(12*(1-MU**2))*(t/b)**2
  return sigma

def case3(t, b):

  sigma = (6*math.pi**2 * E)/(12*(1-MU**2))*(t/b)**2
  return sigma


def testFail():
  """
  increment p until the bridge breaks


  for every iteration of p
     compare the moment at each time and see if its less then the max
  """
```

```
    # global sfd
    broken1 = False
    broken2 = False
    broken3 = False
    broken4 = False
    broken5 = False
    p = 600

    while not broken1 and not broken2 and not broken3 and not broken4 and not broken5:
        buildSFD(565, p)
        # printSFD()

        arr = buildSFD(1265,p)
        # print(arr[676])
        with open('your_file.txt', 'w') as f:
            for item in arr:
                f.write("%s\n" % item)

        buildBMD()
        m_buckle = MFailBuck(1.27)
        m_tension = M_failMatT(30)
        m_compression = M_failMatC(6)
        v_fails = V_fail(4)
        v_bucks = V_buck(520)
        for i in range(L-1):
            if abs(m_buckle[i]) < abs(bmd[i]):
                broken1 = True
                print("BUCKLE FAIL at " + str(i) + " at load" + str(p))
                print(bmd[i])
            if abs(m_tension[i]) < abs(bmd[i]):
                broken2 = True
                print("Tension FAIL at " + str(i)+ " at load" + str(p))
                print(bmd[i])
            if abs(m_compression[i]) < abs(bmd[i]):
                broken3 = True
                print("COMPRESSION FAIL at " + str(i)+ " at load" + str(p))
                print(bmd[i])
            if abs(v_fails[i]) < abs(arr[i]):

                broken4 = True
                print("sheer failure at " + str(i) + " at load " + str(p) + " sheer fail value = " + str(arr[i]))
            if abs(v_bucks[i]) < abs(arr[i]):
                print(arr[i])
                broken5 = True
                print("sheer buckle failure at " + str(i) + " at load " + str(p) + " sheer fail value = " + str(arr[i]))
                pass
        p+=1
def testFailTrain():
    """
    function to test failure load when the train is in the middle of the supports
    incrimentally adds 1 to P until the bridge fails
    functions prints what caused failure

    """
    broken1 = False
    broken2 = False
    broken3 = False
    broken4 = False
    broken5 = False
    p = 30
    # while not broken1 or not broken2 or not broken3:
    while not broken1 and not broken2 and not broken3 and not broken4 and not broken5:
        #for values of train at the right
        # buildSFD(372, p)
        # buildSFD(548,p)
        # buildSFD(712, p)
```

```
        # buildSFD(888,p)
        # buildSFD(1052, p)
        # arr =buildSFD(1228,p)


        # for values at center
        buildSFD(117, p)
        buildSFD(293,p)
        buildSFD(457, p)
        buildSFD(633,p)
        buildSFD(797, p)
        arr =buildSFD(973,p)

        buildBMD()

        m_buckle = MFailBuck(1.27)
        m_tension = M_failMatT(30)
        m_compression = M_failMatC(6)
        v_fails = V_fail(4)
        v_bucks = V_buck(520)

        for i in range(L-1):
            if abs(m_buckle[i]) < abs(bmd[i]):
                broken1 = True
                print("BUCKLE FAIL at " + str(i) + " at load" + str(p))
                print(bmd[i])
            if abs(m_tension[i]) < abs(bmd[i]):
                broken2 = True
                print("Tension FAIL at " + str(i)+ " at load" + str(p))
                print(bmd[i])
            if abs(m_compression[i]) < abs(bmd[i]):
                broken3 = True
                print("COMPRESSION FAIL at " + str(i)+ " at load" + str(p))
                print(bmd[i])
            if abs(v_fails[i]) < abs(arr[i]):
                 print(arr[i])
                 broken4 = True
                 print("sheer failure at " + str(i) + " at load " + str(p) + " sheer fail value = " + str(arr[i]))
            if abs(v_bucks[i]) < abs(arr[i]):
                 print(arr[i])
                 broken5 = True
                 print("sheer buckle failure at " + str(i) + " at load " + str(p) + " sheer fail value = " + str(arr[i]))

        p+=1

def deflection():

    """
    function to return midspan deflection of the bridge based on point loads of 200N

        function uses the areas of the various triangles formed on the BMD to calculate the midspan deflection
        uses similar triangles with tangential deviation from the min moment to the tangenet draw at the begining of the curvutre
diagra,

    returns midspan deflection

    """
    p=200
    buildSFD(550, p)

    arr = buildSFD(1250,p)
    buildBMD()
    I = geometric_properties()[0]
```

```
    mid_point = int((550+510)/2)

    curvatures = []
    buildBMD()
    for i in range(len(bmd)):
        curvatures.append(bmd[i]/(E*I[i]))

    print("10")
    distance_from_max_moment_to_0_moment = 237
    distance_from_0_moment_to_min_moment = 272
    distance_from_min_moment_to_end = 190
    distance_to_max = 550+support_x
    distance_to_mid = 530+support_x_right
    curv_mid = curvatures[mid_point+ support_x]
    curv_max = curvatures[550 + support_x]
    curv_min =
curvatures[550+distance_from_max_moment_to_0_moment+distance_from_0_moment_to_min_moment+support_x]

    delta_c_a = 1/2*(curv_max)*distance_to_max*(distance_from_max_moment_to_0_moment +
distance_from_0_moment_to_min_moment+1/3(distance_to_max))
    delta_c_a += 1/2*(curv_max * distance_from_max_moment_to_0_moment)*(distance_from_0_moment_to_min_moment +
2/3*distance_from_max_moment_to_0_moment)
    delta_c_a+= 1/2*(curv_min)*distance_from_0_moment_to_min_moment*1/3*distance_from_0_moment_to_min_moment

    delta_mid_a = 1/2 *curv_min * distance_to_mid * 1/3*distance_to_mid

    delta_mid = 1/2 *delta_c_a - delta_mid_a

    return delta_mid


def graphs():
    """
    function to generate nessisary graphs
    """
    p=617
    buildSFD(550, p)
    arr = buildSFD(1250,p)
    printSFD()
    buildBMD()
    printBMD()
    m_buckle = MFailBuck(1.27)
    m_tension = M_failMatT(30)
    m_compression = M_failMatC(6)
    v_fails = V_fail(4)
    v_bucks = V_buck(520)
    bmd_momentFails(m_compression, m_tension)
    bmd_momentBuckFail()
    sheerFailSFD(v_fails)
    shearBuck(v_bucks)


def bmd_momentFails(m_compression, m_tensions):
    """
    functon to graph the bmd vs material moment failures
    """
    plt.title("BMD vs Material Moment Failures")
    plt.xlabel("x (mm)")
    plt.ylabel("bending moment (Nmm)")
    zeroliney = [0, 0]
    zerolinex = [0, L]
    plt.plot(zerolinex, zeroliney)

    # plt.plot(*zip(*sorted(buildBMD(xP, P))))
    plt.gca().invert_yaxis()
    plt.plot(bmd)
    plt.plot(m_compression, label = "Matboard Compression Failure")
```

```
    plt.plot(m_tensions, label = "Matboard Tension Failure")
    plt.legend()
    plt.show()

def bmd_momentBuckFail():
    """
    functon to graph the bmd vs material moment failures
    """
    plt.title("BMD vs Material Buckling Failures")
    plt.xlabel("x (mm)")
    plt.ylabel("bending moment (Nmm)")
    zeroliney = [0, 0]
    zerolinex = [0, L]
    plt.plot(zerolinex, zeroliney)

    # plt.plot(*zip(*sorted(buildBMD(xP, P))))
    plt.gca().invert_yaxis()
    plt.plot(bmd)
    plt.plot(sigmas1, label = "Mid Flange Buckling")
    plt.plot(sigmas2, label = "Side flange buckling")
    plt.plot(sigmas3, label = "Web compression buckling")
    plt.legend()
    plt.show()

def sheerFailSFD(shear):
    """
    function to graph SFD vs shear failures
    """
    plt.title("SFD vs Material Shear Failures")
    plt.xlabel("x (mm)")
    plt.ylabel("shear force (N)")
    arr = []
    for i in range(len(sfd)):
        arr.append(sfd[i][0])
    zeroliney = [0, 0]
    zerolinex = [0, L]
    plt.plot(zerolinex, zeroliney)
    plt.plot(arr)
    plt.plot(shear, label = "Matboard Shear Failrue", color = "b")
    negative_shear = []
    for i in shear:
        negative_shear.append(i*-1)
    plt.plot(negative_shear, color = "b")
    plt.legend()
    plt.show()

def shearBuck(buck):
    """
    functions to graph sfd vs shear buckling failure
    """
    plt.title("SFD vs  Shear Buckling Failures")
    plt.xlabel("x (mm)")
    plt.ylabel("shear force (N)")
    arr = []
    for i in range(len(sfd)):
        arr.append(sfd[i][0])
    zeroliney = [0, 0]
    zerolinex = [0, L]
    plt.plot(zerolinex, zeroliney)
    plt.plot(arr)
    plt.plot(buck, label = "Web Shear Buckle Failrue", color = "b")
    negative_buck = []
    for i in buck:
        negative_buck.append(i*-1)
    plt.plot(negative_buck, color = "b")
    plt.legend()
    plt.show()
```

```
def FOStrain():
    """
    function to calculate FOS between the shear forces and internal moments

    the FOS at each point of the bridge is calculated by dividing the fail causing failure by the actual expierenced values
    the minimum of these values is then used as the FOS

        returns:
            min(fos_m)(float): FOS for the moments
            min(fos_v)(float): FOS for the shear forces
    """
    global sfd
     # for values at center
    p=400/6
    buildSFD(117, p)
    buildSFD(293,p)
    buildSFD(457, p)
    buildSFD(633,p)
    buildSFD(797, p)
    arr =buildSFD(973,p)
    buildBMD()
    m_fail_comp = M_failMatC(6)
    shear_fail = V_fail(4)
    m_fail_buck = MFailBuck(1.27)
    fos_m = []

    fos_v = []
    for i in range(len(bmd)):
        if(bmd[i] != 0):
            fos_m.append(m_fail_comp[i] / bmd[i])
            fos_m.append(m_fail_buck[i]/bmd[i])
        if(arr[i] != 0):
            fos_v.append(shear_fail[i]/abs(arr[i]))
    return min(fos_m), min(fos_v)


def glue_tabs(Tau):
    """
    returns the shear needed to break the glue

        parameters:
            Tau(float): the shear strength of the glue
        returns:
            glue_tab_values(list): the shear needed to break the glue at anypoint along the bridge
    """
    I = geometric_properties()[0]
    Q = geometric_properties()[2]
    centroid = geometric_properties()[1]
    bottomthickness = get_properties()[4]
    height = get_properties()[0]
    topthickness = get_properties()[3]
    widthbottom = get_properties()[2]
    widthtop = get_properties()[1]
    rightthickness = get_properties()[5]
    leftthickness = get_properties()[6]
    glue_tabs_values = []

    for i in range(L):
        if 0 <= centroid[i] < bottomthickness[i]:
            centroidlocation = "bottom"
        elif bottomthickness[i] <= centroid[i] < height[i] - topthickness[i]:
            centroidlocation = "middle"
        elif height[i] - topthickness[i] <= centroid[i] < height[i]:
            centroidlocation = "top"
```

```
        if centroidlocation == "bottom":
            b = widthbottom[i]
        elif centroidlocation == "middle":
            b = rightthickness[i] + leftthickness[i]
        elif centroidlocation == "top":
            b = widthtop[i]

        glue_tab_force = (Tau * I[i] * b)/Q[i]

        glue_tabs_values.append(glue_tab_force)

    return glue_tabs_values



if __name__ == "__main__":
    # buildSFD(100,50)
    # printSFD()
    # buildBMD()
    # printBMD()
    # init()
    # testFail()
    # testFailTrain()
    # graphs()
    # print(FOStrain())
    print(deflection())
```