```
import math
import numpy as np
from pyDOE import lhs
import math
import numpy as np

def Euler(c, E, Lp):
    return c * (math.pi**2 * E) / Lp**2

def Johnson(c, E, Lp, y_sig):
    return y_sig * (1 - (y_sig * Lp**2)/(4*c*math.pi**2 * E))

def Gerard(kc, E, v, t, b):
    return ((math.pi**2 * kc * E) / (12*(1-v**2))) * (t/b)**2

# Objective Function
def f(tskin, tstiff, nstiff, wstiff):
    # Given Constants
    L = 90
    hstiff = .1
    A1 = wstiff * tstiff
    A2 = abs(tstiff * (hstiff-tstiff))

    cen_1x = wstiff/2
    cen_1y = tstiff/2

    cen_2x = tstiff/2
    cen_2y = (hstiff-tstiff)/2 + tstiff

    # print(hstiff, wstiff, tstiff, tskin, nstiff)
    # print("Areas:", A1, A2)
    y_tot = (wstiff*tstiff * tstiff/2 + (hstiff-tstiff)*tstiff * ((hstiff-tstiff)/2
+ tstiff)) / (A1 + A2)
    x_tot = (wstiff*tstiff * wstiff/2 + (hstiff-tstiff)*tstiff * (tstiff/2))/ (A1 +
A2)
    # print("y_tot = ", y_tot)
    # print("x_tot = ", x_tot)

    xh1 = y_tot - cen_1y
    xh2 = cen_2y - y_tot
    Ixx1 = (wstiff*tstiff**3)/12 + A1*xh1**2
    Ixx2 = (tstiff*(hstiff-tstiff)**3)/12 + A2*xh2**2
    Ixx_tot = Ixx1 + Ixx2

    # print("Ixx1 = ", Ixx1)
    # print("Ixx2 = ", Ixx2)
    # print("Ixx_tot = ", Ixx_tot)

    # yh1 = x_tot - cen_1x
    # yh2 = cen_2x - x_tot
```

```
    # Iyy1 = (tstiff*wstiff**3)/12 + A1*yh1**2
    # Iyy2 = ((hstiff-tstiff)*tstiff**3)/12 + A2*yh2**2
    # Iyy_tot = Iyy1 + Iyy2

    # print("Iyy1 = ", Iyy1)
    # print("Iyy2 = ", Iyy2)
    # print("Iyy_tot = ", Iyy_tot)

    # Find Smallest Inertia
    min_I = Ixx_tot

    # Find slenderness ratio
    c = 4
    E = 11 * 10**6
    y_sig = 50 * 10**3
    Lp = math.pi * math.sqrt(2 * c * E/y_sig)
    # print("\nSlenderness Ratio at E=J: ", Lp)

    # Solve for radius of gyration
    rho = math.sqrt(min_I/(A1 + A2))
    # print("Radius of Gyration: ", rho)

    # Find column slenderness ratio
    Lp_col = L / rho
    # print("Column slenderness ratio: ", Lp_col)

    colCr = 0

    # Compare slenderness ratio to column slenderness ratio and find crit buckling
stress
    if(Lp_col > Lp):
        # print("\nUsing Euler's Solution to find critical buckling stress")
        colCr = Euler(c, E, Lp_col)
    else:
        # print("\nUsing Johnson's Solution to find critical buckling stress")
        colCr = Johnson(c, E, Lp_col, y_sig)

    # print("Critical buckling stress: ", colCr)

    # For the thin plate
    Kc = 7.2
    v = .3
    wDomain = 60 / nstiff

    plateCr = Gerard(Kc, E, v, tskin, wDomain)
    # print("\nUsing Gerard's Solution to find Plate critical buckling stress")
    # print("Critical Buckling Stress: ", plateCr)

    # Solving for Fstiff and Fskin
```

```
    Astiff = A1 + A2
    Askin = wDomain * tskin

    Fstiff = 40 * wDomain * Astiff / (Astiff + Askin)
    # print("\nF_stiff: ", Fstiff)

    Fskin = 40 * wDomain * Askin / (Askin + Astiff)
    # print("F_skin: ", Fskin)

    # Turn Critical Stress into Critical Load
    # Column Stress to load
    colLoad = colCr * Astiff
    # print("Critical Load of Stiffener: ", colLoad)

    plateLoad = plateCr * Askin
    #print("Critical Load of Plate: ", plateLoad)

    colMass = L * Astiff * .1
    plateMass = L * Askin * .1
    totalMass = colMass * nstiff + plateMass

    # Actual Stress
    StressStiff = Fstiff / Astiff
    StressSkin = Fskin / Askin


    minStress = min(StressSkin, StressStiff)

    return minStress, totalMass, colCr, plateCr




############### Particle Swarm Function ##################

def PSO(swarm_size, tskin, tstiff, nstiff, wstiff, numIter):
    bounds = [tskin, tstiff, nstiff, wstiff]

    # Initialize the Particles and their locations using Latin Hyper Square
    swarm = lhs(len(bounds), samples=swarm_size, criterion='center')
    for j in range(len(bounds)):
        swarm[:, j] = swarm[:, j] * (bounds[j][1] - bounds[j][0]) + bounds[j][0]

    # Initialize Particle Velocities (Initial Velocity is 0)
    velocities = np.empty([10,4])

    pbest_pos = swarm.copy()

    pbest_cost = []
    data = []
```

```
    for i in swarm:
        output = f(i[0],i[1],i[2],i[3])
        minStress = output[0]
        colCr = output[2]
        plateCr = output[3]

        data.append(output)
        if(minStress > colCr or minStress > plateCr):
            pbest_cost.append(output[1] * 100000)
        else:
            pbest_cost.append(output[1])


    # print(pbest_cost)

    # Initialize global best position and fbest
    gbest_pos = pbest_pos[pbest_cost.index(min(pbest_cost))]
    gbest_cost = min(pbest_cost)
    # print("Min Mass Global Position: ", gbest_pos)
    # print("Min Mass Global: ", gbest_cost)
    # Iterate throught the swarm
    a = .7
    A = .5
    B = .5

    for i in range(numIter):

        for j in range(swarm_size):
            for k in range(len(bounds)):

                # Formula from lecture notes.
                # i is greater than len of velocities
                Xi = swarm[j]
                Vi = velocities[j]

                # Create the new velocity for each particle through the equation
given in lecture
                Vi_new = .7*Vi + .5*np.random.rand()*(pbest_pos[j] - Xi) +
.5*np.random.rand()*(gbest_pos - Xi)
                Vi_new[2] = round(Vi_new[2])
                velocities[j] = Vi_new
                Xi_new = Xi + Vi_new
                Xi_new[2] = round(Xi_new[2])
                # Update the position of the particle
                swarm[j] = Xi_new


                # Check if the new position violates lower and upper bounds
                if(swarm[j][k] < bounds[k][0]):
                    swarm[j][k] = bounds[k][0]
```

```
                    velocities[j][k] = 0
                elif(swarm[j][k] > bounds[k][1]):
                    swarm[j][k] = bounds[k][1]
                    velocities[j][k] = 0

        # Compare the current position cost to pbest and update for each particle
        for j in range(swarm_size):
            # print('swarm')
            # print(swarm[j])
            fstress, fmass, fcolCr, fplateCr = f(swarm[j][0],swarm[j][1],
swarm[j][2], swarm[j][3] )
            # Check Constraint
            if (fmass < pbest_cost[j] and (fstress < fcolCr and fstress <
fplateCr)):
                pbest_cost[j] = fmass
                pbest_pos[j] = swarm[j]

        # Update the global best position and cost
        minCost = min(pbest_cost)
        if(minCost < gbest_cost):
            gbest_cost = minCost
            gbest_pos = pbest_pos[pbest_cost.index(gbest_cost)]

    return gbest_pos, gbest_cost

################################ Main ########################

swarm_size = 10
tskin = (0.05, 0.15)
tstiff = (0.05, 0.16)
nstiff = (5, 15)
hstiff = .1
wstiff = (0.05, 0.15)
numIter = 2000
best_pos, best_cost = PSO(swarm_size, tskin, tstiff, nstiff, wstiff, numIter)
print("Initial DV bounds")
print("tskin: ", tskin)
print("tstiff: ", tstiff)
print("nstiff: ", nstiff)
print("hstiff: ", hstiff)
print("wstiff: ", wstiff)
print("Optimized DV values: ", best_pos)
print("Minimized Mass: ", best_cost)
    # cost is the cost of the function and in this case is stress? or mass
```