Assignment-Hermite Polynomials based solution

Code

```python
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib as mpl
import warnings
warnings.filterwarnings("ignore")
mpl.rcParams['figure.max_open_warning'] = 0

output_folder = "Plots"
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

def bc_label(bc):
    if bc == 'SSSS':
        return "All Edges Simpb-Supported"
    elif bc == 'CFFF':
        return " Clamped, Others Free"

def plot_surface_disp(X, Y, Z, p, bc, a, b):
    Z_mm = Z * 1000.0
    fig = plt.figure(figsize=(12,9))
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(X, Y, Z_mm, cmap='viridis', edgecolor='none')
    ax.set_title(f'Deflection Surface, Boundary Condition: {bc_label(bc)}, p={p}')
    ax.set_xlabel('x (m)')
    ax.set_ylabel('y (m)')
    ax.set_zlabel('Deflection (mm)')
    ax.set_xlim(0, a)
    ax.set_ylim(0, b)
    ax.view_init(elev=30, azim=140)
    cbar = fig.colorbar(surf)
    cbar.set_label("Deflection (mm)")
    filename = os.path.join(output_folder, f"deflection_surface_p{p}_{bc}.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def plot_contour_disp(X, Y, Z, p, bc, a, b):
    Z_mm = Z * 1000.0
    fig = plt.figure(figsize=(12,9))
    cp = plt.contourf(X, Y, Z_mm, 30, cmap='jet')
    cbar = plt.colorbar(cp)
    cbar.set_label("Deflection (mm)")
    plt.title(f'Deflection Contour, Boundary Condition: {bc_label(bc)}, p={p}')
    plt.xlabel('x (m)')
    plt.ylabel('y (m)')
    plt.axis('equal')
    plt.axis([0, a, 0, b])
    filename = os.path.join(output_folder, f"deflection_contour_p{p}_{bc}.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def plot_vm_contour(X, Y, sigma_vm, p, bc, a, b):
    fig = plt.figure(figsize=(12,9))
    cp = plt.contourf(X, Y, sigma_vm, 20, cmap='jet')
    plt.colorbar(cp)
    plt.title(f'Von Mises Stress, Boundary Condition: {bc_label(bc)}, p={p}')
    plt.xlabel('x (m)')
    plt.ylabel('y (m)')
    plt.axis('equal')
    plt.axis([0, a, 0, b])
    filename = os.path.join(output_folder, f"von_mises_p{p}_{bc}.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def plot_cubic_hermite_functions():
    xi = np.linspace(0, 1, 200)
    H1_vals = np.array([H1(x)[0] for x in xi])
    H2_vals = np.array([H2(x)[0] for x in xi])
    H3_vals = np.array([H3(x)[0] for x in xi])
    H4_vals = np.array([H4(x)[0] for x in xi])

    fig = plt.figure(figsize=(12,9))
    plt.plot(xi, H1_vals, label='H1')
    plt.plot(xi, H2_vals, label='H2')
    plt.plot(xi, H3_vals, label='H3')
    plt.plot(xi, H4_vals, label='H4')
    plt.xlabel('')
    plt.ylabel('Shape Function Value')
    plt.title('Cubic Hermite Shape Functions')
    plt.legend()
    plt.grid(True)
    filename = os.path.join(output_folder, "cubic_hermite_functions.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def plot_through_thickness_x(z_vals, sig_x, sig_y, sig_xy, p, bc):
    fig = plt.figure(figsize=(10,8))
    plt.plot(sig_x, z_vals, linewidth=2)
    plt.grid(True)
    plt.xlabel(r'$\sigma_{xx}$')
    plt.ylabel('z (m)')
    plt.title(r'$\sigma_{xx}$ vs z')
    plt.suptitle(f'Through-thickness stresses at center, Boundary Condition: {bc_label(bc)}, p={p}')
    filename = os.path.join(output_folder, f"through_thickness_x_p{p}_{bc}.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def plot_through_thickness_y(z_vals, sig_x, sig_y, sig_xy, p, bc):
    fig = plt.figure(figsize=(10,8))
    plt.plot(sig_y, z_vals, linewidth=2)
    plt.grid(True)
    plt.xlabel(r'$\sigma_{yy}$')
    plt.ylabel('z (m)')
    plt.title(r'$\sigma_{yy}$ vs z')
    plt.suptitle(f'Through-thickness stresses at center, Boundary Condition: {bc_label(bc)}, p={p}')
    filename = os.path.join(output_folder, f"through_thickness_y_p{p}_{bc}.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def plot_through_thickness_xy(z_vals, sig_x, sig_y, sig_xy, p, bc):
    fig = plt.figure(figsize=(10,8))
    plt.plot(sig_xy, z_vals, linewidth=2)
    plt.grid(True)
    plt.xlabel(r'$\sigma_{xy}$')
    plt.ylabel('z (m)')
    plt.title(r'$\sigma_{xy}$ vs z')
    plt.suptitle(f'Through-thickness stresses at center, Boundary Condition: {bc_label(bc)}, p={p}')
    filename = os.path.join(output_folder, f"through_thickness_xy_p{p}_{bc}.png")
    plt.savefig(filename, dpi=300, bbox_inches='tight')
    plt.close(fig)

def assemble_plate_system(p, a, b, q, D, nu, bc_type):
    n_nodes = p + 1
    total_dof = n_nodes * n_nodes
    K = np.zeros((total_dof, total_dof))
    F = np.zeros(total_dof)

    gauss_points_x, gauss_weights_x = gauss_quadrature(10, 0, a)
    gauss_points_y, gauss_weights_y = gauss_quadrature(10, 0, b)

    for ix, (x, wx) in enumerate(zip(gauss_points_x, gauss_weights_x)):
        phi_x = np.zeros(n_nodes)
        d2phi_x = np.zeros(n_nodes)
        for i in range(1, n_nodes+1):
            val, _, d2val = hermite_shape(p, i, x, a)
            phi_x[i-1] = val
            d2phi_x[i-1] = d2val

        for iy, (y, wy) in enumerate(zip(gauss_points_y, gauss_weights_y)):
            weight = wx * wy
            phi_y = np.zeros(n_nodes)
            d2phi_y = np.zeros(n_nodes)
            for j in range(1, n_nodes+1):
                val, _, d2val = hermite_shape(p, j, y, b)
                phi_y[j-1] = val
                d2phi_y[j-1] = d2val

            for i in range(1, n_nodes+1):
                for j in range(1, n_nodes+1):
                    idx = (i-1)*n_nodes + (j-1)
                    F[idx] += q * (phi_x[i-1] * phi_y[j-1]) * weight
                    phi_xx = d2phi_x[i-1] * phi_y[j-1]
                    phi_yy = phi_x[i-1] * d2phi_y[j-1]
                    phi_xy = first_derivative(p, i, x, a) * first_derivative(p, j, y, b)

                    for m in range(1, n_nodes+1):
                        for n in range(1, n_nodes+1):
                            jdx = (m-1)*n_nodes + (n-1)
                            phi_xx_m = d2phi_x[m-1] * phi_y[n-1]
                            phi_yy_m = phi_x[m-1] * d2phi_y[n-1]
                            phi_xy_m = first_derivative(p, m, x, a) * first_derivative(p, n, y, b)
                            integrand = (phi_xx * phi_xx_m + phi_yy * phi_yy_m +
                                         2*(1-nu)*phi_xy * phi_xy_m +
                                         nu*(phi_xx * phi_yy_m + phi_yy * phi_xx_m))
                            K[idx, jdx] += D * integrand * weight

    constrained = determine_boundary_indices(p, bc_type)
    for index in constrained:
        K[index, :] = 0
        K[:, index] = 0
        K[index, index] = 1
        F[index] = 0

    return K, F

def compute_plate_deflection(p, coeff, a, b):
    n_points = 21
    x_coords = np.linspace(0, a, n_points)
    y_coords = np.linspace(0, b, n_points)
    disp = np.zeros((n_points, n_points))
    n_nodes = p + 1

    for i, x in enumerate(x_coords):
        phi_x = np.array([hermite_shape(p, i+1, x, a)[0] for i in range(n_nodes)])
        for j, y in enumerate(y_coords):
            phi_y = np.array([hermite_shape(p, j+1, y, b)[0] for j in range(n_nodes)])
            summ = 0
            index = 0
            for ix in range(n_nodes):
                for iy in range(n_nodes):
                    summ += coeff[index] * phi_x[ix] * phi_y[iy]
                    index += 1
            disp[j, i] = summ
    X, Y = np.meshgrid(x_coords, y_coords)
    return X, Y, disp

def evaluate_derivatives(p, coeff, x, y, a, b):
    n_nodes = p + 1
    val = 0
    d2x = 0
    d2y = 0
    dxy = 0

    phi_x = np.zeros(n_nodes)
    dphi_x = np.zeros(n_nodes)
    d2phi_x = np.zeros(n_nodes)
    for i in range(1, n_nodes+1):
        v, dv, d2v = hermite_shape(p, i, x, a)
        phi_x[i-1] = v
        dphi_x[i-1] = dv
        d2phi_x[i-1] = d2v

    phi_y = np.zeros(n_nodes)
    dphi_y = np.zeros(n_nodes)
    d2phi_y = np.zeros(n_nodes)
    for j in range(1, n_nodes+1):
        v, dv, d2v = hermite_shape(p, j, y, b)
        phi_y[j-1] = v
        dphi_y[j-1] = dv
        d2phi_y[j-1] = d2v

    index = 0
    for i in range(n_nodes):
        for j in range(n_nodes):
            c = coeff[index]
            val += c * (phi_x[i] * phi_y[j])
            d2x += c * (d2phi_x[i] * phi_y[j])
            d2y += c * (phi_x[i] * d2phi_y[j])
            dxy += c * (dphi_x[i] * dphi_y[j])
            index += 1

    return val, d2x, d2y, dxy

def hermite_shape(p, local_index, X, A):
    if p == 3:
        return cubic_shape(local_index, X, A)
    elif p == 4:
        if local_index <= 4:
            return cubic_shape(local_index, X, A)
        else:
            return pob2_shape(X, A)
    elif p == 5:
        if local_index <= 4:
            return cubic_shape(local_index, X, A)
        elif local_index == 5:
            return pob2_shape(X, A)
        else:
            return pob3_shape(X, A)

def cubic_shape(index, x, A):
    xi = x / A
    if index == 1:
        ref, dref, d2ref = H1(xi)
        value = ref
        dvalue = dref / A
        d2value = d2ref / (A**2)
    elif index == 2:
        ref, dref, d2ref = H2(xi)
        value = A * ref
        dvalue = dref
        d2value = d2ref / A
    elif index == 3:
        ref, dref, d2ref = H3(xi)
        value = ref
        dvalue = dref / A
        d2value = d2ref / (A**2)
    elif index == 4:
        ref, dref, d2ref = H4(xi)
        value = A * ref
        dvalue = dref
        d2value = d2ref / A
    return value, dvalue, d2value

def H1(xi):
    v = 1 - 3*xi**2 + 2*xi**3
    dv = -6*xi + 6*xi**2
    d2v = -6 + 12*xi
    return v, dv, d2v

def H2(xi):
    v = xi*(1 - 2*xi + xi**2)
    dv = (1 - 2*xi + xi**2) + xi*(-2 + 2*xi)
    d2v = -4 + 6*xi
    return v, dv, d2v

def H3(xi):
    v = 3*xi**2 - 2*xi**3
    dv = 6*xi - 6*xi**2
    d2v = 6 - 12*xi
    return v, dv, d2v

def H4(xi):
    v = xi**2*(xi - 1)
    dv = 3*xi**2 - 2*xi
    d2v = 6*xi - 2
    return v, dv, d2v

def pob2_shape(x, A):
    f1 = x**2;  df1 = 2*x;  d2f1 = 2
    f2 = (A - x)**2; df2 = -2*(A - x); d2f2 = 2
    scale = 1 / (A**4)
    v = scale * (f1 * f2)
    dv = scale * (df1 * f2 + f1 * df2)
    d2v = scale * (d2f1 * f2 + 2*df1*df2 + f1 * d2f2)
    return v, dv, d2v

def pob3_shape(x, A):
    f1 = x**3;  df1 = 3*x**2;  d2f1 = 6*x
    f2 = (A - x)**3; df2 = -3*(A - x)**2; d2f2 = 6*(A - x)
    scale = 1 / (A**6)
    prod = f1 * f2
    dprod = df1 * f2 + f1 * df2
    d2prod = d2f1 * f2 + 2*df1*df2 + f1*d2f2
    v = scale * prod
    dv = scale * dprod
    d2v = scale * d2prod
    return v, dv, d2v

def first_derivative(p, local_index, x, A):
    _, d_val, _ = hermite_shape(p, local_index, x, A)
    return d_val

def determine_boundary_indices(p, bc_type):
    n_nodes = p + 1
    indices = []
    if bc_type == 'SSSS':
        remove_x = [1, 3]
        remove_y = [1, 3]
        for i in range(1, n_nodes+1):
            for j in range(1, n_nodes+1):
                if i in remove_x or j in remove_y:
                    indices.append((i-1)*n_nodes + (j-1))
    elif bc_type == 'CFFF':
        for i in range(1, n_nodes+1):
            for j in range(3, n_nodes+1):
                indices.append((i-1)*n_nodes + (j-1))
    return sorted(set(indices))

def gauss_quadrature(n, a, b):
    if n == 1:
        points  = np.array([0])
        weights = np.array([2])
    elif n == 2:
        points  = np.array([-1/np.sqrt(3), 1/np.sqrt(3)])
        weights = np.array([1, 1])
    elif n == 3:
        points  = np.array([-np.sqrt(3/5), 0, np.sqrt(3/5)])
        weights = np.array([0.55555, 0.88889, 0.55555])
    elif n == 4:
        points  = np.array([-0.86111, -0.33990, 0.33990, 0.86111])
        weights = np.array([0.34780, 0.65210, 0.65210, 0.34780])
    elif n == 5:
        points  = np.array([-0.90610, -0.53840,  0,  0.53840,  0.90610])
        weights = np.array([0.23690, 0.47860, 0.56880, 0.47860, 0.23690])
    elif n == 6:
        points  = np.array([-0.93247, -0.66121, -0.23862,
                             0.23862,  0.66121,  0.93247])
        weights = np.array([0.17132, 0.36076, 0.46791, 0.46791, 0.36076, 0.17132])
    elif n == 7:
        points  = np.array([-0.94911, -0.74153, -0.40585,
                             0, 0.40585,  0.74153,  0.94911])
        weights = np.array([0.12948, 0.27971, 0.38183,
                             0.41796, 0.38183, 0.27971, 0.12948])
    elif n == 8:
        points  = np.array([-0.96029, -0.79667, -0.52553, -0.18343,
                             0.18343, 0.52553,  0.79667,  0.96029])
        weights = np.array([0.10123, 0.22238, 0.31371, 0.36268,
                             0.36268, 0.31371, 0.22238, 0.10123])
    elif n == 9:
        points  = np.array([-0.96816, -0.83603, -0.61337,
                            -0.32425, 0, 0.32425,  0.61337,  0.83603, 0.96816])
        weights = np.array([0.08127, 0.18065, 0.26061,
                             0.31235, 0.33024, 0.31235, 0.26061, 0.18065, 0.08127])
    elif n == 10:
        points  = np.array([-0.97391, -0.86506, -0.67941, -0.43340,
                            -0.14887,  0.14887,  0.43340,  0.67941,
                             0.86506,  0.97391])
        weights = np.array([0.06667, 0.14945, 0.21909, 0.26927,
                             0.29552, 0.29552, 0.26927, 0.21909,
                             0.14945, 0.06667])
    mid = 0.5*(a+b)
    half_length = 0.5*(b-a)
    transformed_points = mid + half_length * points
    transformed_weights = half_length * weights
    return transformed_points, transformed_weights

def main():
    E = 200e9
    neu = 0.3
    h = 4e-3
    a = 0.5
    b = 0.5
    qo = 1000
    D = E * h**3 / (12*(1 - neu**2))

    p_values = [3, 4, 5]
    boundary_conditions = ['SSSS', 'CFFF']
    yield_stress = 450e6

    for p in p_values:
        for bc in boundary_conditions:
            stiffness, force = assemble_plate_system(p, a, b, qo, D, neu, bc)
            coeff = np.linalg.solve(stiffness, force)

            X, Y, displacement = compute_plate_deflection(p, coeff, a, b)
            plot_surface_disp(X, Y, displacement, p, bc, a, b)
            plot_contour_disp(X, Y, displacement, p, bc, a, b)

            Nx, Ny = 31, 31
            x_grid = np.linspace(0, a, Nx)
            y_grid = np.linspace(0, b, Ny)
            X_grid, Y_grid = np.meshgrid(x_grid, y_grid)
            top_z = h/2

            vm_stress = np.zeros((Ny, Nx))
            max_vm = -np.inf
            max_xy = (0, 0)

            for ix, x in enumerate(x_grid):
                for iy, y in enumerate(y_grid):
                    _, zxx, zyy, zxy = evaluate_derivatives(p, coeff, x, y, a, b)
                    sig_x = -(E/(1-neu**2)) * top_z * (zxx + neu*zyy)
                    sig_y = -(E/(1-neu**2)) * top_z * (zyy + neu*zxx)
                    sig_xy = -(E/(1+neu)) * top_z * zxy
                    vm = np.sqrt(0.5*((sig_x - sig_y)**2 + sig_x**2 + sig_y**2) + 3*(sig_xy**2))
                    vm_stress[iy, ix] = vm
                    if vm > max_vm:
                        max_vm = vm
                        max_xy = (x, y)

            plot_vm_contour(X_grid, Y_grid, vm_stress, p, bc, a, b)
            q_yield = (yield_stress / max_vm) * qo
            print(f'p={p}, bc={bc}: Max VM Stress={max_vm:.6e} at ({max_xy[0]:.2f},{max_xy[1]:.2f})')
            print(f'p={p}, bc={bc}: q_yield={q_yield:.2e} N/m²')

            strain_energy = 0.5 * coeff.T @ stiffness @ coeff
            print(f'p={p}, bc={bc}: Strain Energy = {strain_energy:.2e} J\n')

            center_x, center_y = a/2, b/2
            z_positions = np.linspace(-h/2, h/2, 51)
            _, zxx_c, zyy_c, zxy_c = evaluate_derivatives(p, coeff, center_x, center_y, a, b)
            sig_x_profile = np.array([-(E/(1-neu**2)) * z * (zxx_c + neu*zyy_c) for z in z_positions])
            sig_y_profile = np.array([-(E/(1-neu**2)) * z * (zyy_c + neu*zxx_c) for z in z_positions])
            sig_xy_profile = np.array([-(E/(1+neu)) * z * zxy_c for z in z_positions])

            plot_through_thickness_x(z_positions, sig_x_profile, sig_y_profile, sig_xy_profile, p, bc)
            plot_through_thickness_y(z_positions, sig_x_profile, sig_y_profile, sig_xy_profile, p, bc)
            plot_through_thickness_xy(z_positions, sig_x_profile, sig_y_profile, sig_xy_profile, p, bc)

    plot_cubic_hermite_functions()

if __name__ == "__main__":
    main()
    plt.show()
```

Output

```
p=3, bc=SSSS: Max VM Stress=6.458288e+06 at (0.50,0.50)
p=3, bc=SSSS: q_yield=6.97e+04 N/m²
p=3, bc=SSSS: Strain Energy = 1.05e-02 J

p=3, bc=CFFF: Max VM Stress=3.548457e+07 at (0.25,0.50)
p=3, bc=CFFF: q_yield=1.27e+04 N/m²
p=3, bc=CFFF: Strain Energy = 3.26e-01 J

p=4, bc=SSSS: Max VM Stress=5.338571e+06 at (0.00,0.00)
p=4, bc=SSSS: q_yield=8.43e+04 N/m²
p=4, bc=SSSS: Strain Energy = 1.13e-02 J

p=4, bc=CFFF: Max VM Stress=3.528314e+07 at (0.25,0.50)
p=4, bc=CFFF: q_yield=1.28e+04 N/m²
p=4, bc=CFFF: Strain Energy = 3.27e-01 J

p=5, bc=SSSS: Max VM Stress=5.289816e+06 at (0.00,0.00)
p=5, bc=SSSS: q_yield=8.51e+04 N/m²
p=5, bc=SSSS: Strain Energy = 1.13e-02 J

p=5, bc=CFFF: Max VM Stress=3.532994e+07 at (0.25,0.50)
p=5, bc=CFFF: q_yield=1.27e+04 N/m²
p=5, bc=CFFF: Strain Energy = 3.27e-01 J
```