

Assignment-Hermite Polynomials based solution

Code

```
1 import os
2 import sys
3 import numpy as np
4 import pandas as pd
5 import matplotlib.pyplot as plt
6 from mpl_toolkits.mplot3d import Axes3D
7 import matplotlib as mpl
8 import warnings
9 warnings.filterwarnings("ignore")
10 mpl.rcParams['figure.max_open_warning'] = 0
11
12 output_folder = "Plots"
13 if not os.path.exists(output_folder):
14     os.makedirs(output_folder)
15
16 def bc_label(bc):
17     if bc == 'SSSS':
18         return "All Edges Simpb-Supported"
19     elif bc == 'CFFF':
20         return "Clamped, Others Free"
21
22 def plot_surface_disp(X, Y, Z, p, bc, a, b):
23     Z_mm = Z * 1000.0
24     fig = plt.figure(figsize=(12,9))
25     ax = fig.add_subplot(111, projection='3d')
26     surf = ax.plot_surface(X, Y, Z_mm, cmap='viridis', edgecolor='none')
27     ax.set_title(f'Deflection Surface, Boundary Condition: {bc_label(bc)}',
28                 p='{p}')
29     ax.set_xlabel('x (m)')
30     ax.set_ylabel('y (m)')
31     ax.set_zlabel('Deflection (mm)')
32     ax.set_xlim(0, a)
33     ax.set_ylim(0, b)
34     ax.view_init(elev=30, azim=140)
35     cbar = fig.colorbar(surf)
36     cbar.set_label("Deflection (mm)")
37     filename = os.path.join(output_folder,
38                             f'deflection_surface_p{p}_{bc}.png')
39     plt.savefig(filename, dpi=300, bbox_inches='tight')
40     plt.close(fig)
41
42 def plot_contour_disp(X, Y, Z, p, bc, a, b):
43     Z_mm = Z * 1000.0
44     fig = plt.figure(figsize=(12,9))
45     cp = plt.contourf(X, Y, Z_mm, cmap='jet')
46     cbar = plt.colorbar(cp)
47     cbar.set_label("Deflection (mm)")
48     plt.title(f'Deflection Contour, Boundary Condition: {bc_label(bc)}',
49              p='{p}')
50     plt.xlabel('x (m)')
51     plt.ylabel('y (m)')
52     plt.axis('equal')
53     plt.axis([0, a, 0, b])
54     filename = os.path.join(output_folder,
55                             f'deflection_contour_p{p}_{bc}.png')
56     plt.savefig(filename, dpi=300, bbox_inches='tight')
57     plt.close(fig)
58
59 def plot_vm_contour(X, Y, sigma_vm, p, bc, a, b):
60     fig = plt.figure(figsize=(12,9))
61     cp = plt.contourf(X, Y, sigma_vm, 20, cmap='jet')
62     plt.colorbar(cp)
63     plt.title(f'Von Mises Stress, Boundary Condition: {bc_label(bc)}',
64              p='{p}')
65     plt.xlabel('x (m)')
66     plt.ylabel('y (m)')
67     plt.axis('equal')
68     plt.axis([0, a, 0, b])
69     filename = os.path.join(output_folder, f"von_mises_p{p}_{bc}.png")
70     plt.savefig(filename, dpi=300, bbox_inches='tight')
71     plt.close(fig)
72
73 def plot_cubic_hermite_functions():
74     xi = np.linspace(0, 1, 200)
75     H1_vals = np.array([H1(x)[0] for x in xi])
76     H2_vals = np.array([H2(x)[0] for x in xi])
77     H3_vals = np.array([H3(x)[0] for x in xi])
78     H4_vals = np.array([H4(x)[0] for x in xi])
79
80     fig = plt.figure(figsize=(12,9))
81     plt.plot(xi, H1_vals, label='H1')
82     plt.plot(xi, H2_vals, label='H2')
83     plt.plot(xi, H3_vals, label='H3')
84     plt.plot(xi, H4_vals, label='H4')
85     plt.xlabel('')
86     plt.ylabel('Shape Function Value')
87     plt.title('Cubic Hermite Shape Functions')
88     plt.legend()
89     plt.grid(True)
90     filename = os.path.join(output_folder, "cubic_hermite_functions.png")
91     plt.savefig(filename, dpi=300, bbox_inches='tight')
92     plt.close(fig)
93
94 def plot_through_thickness_x(z_vals, sig_x, sig_y, sig_xy, p, bc):
95     fig = plt.figure(figsize=(10,8))
96     plt.plot(sig_x, z_vals, linewidth=2)
97     plt.grid(True)
98     plt.xlabel(r'$\sigma_x$')
99     plt.ylabel('z (m)')
100     plt.title(r'$\sigma_x$ vs z')
101     plt.suptitle(f'Through-thickness stresses at center, Boundary Condition: {bc_label(bc)}', p='{p}')
102     filename = os.path.join(output_folder,
103                             f"through_thickness_x_p{p}_{bc}.png")
104     plt.savefig(filename, dpi=300, bbox_inches='tight')
105     plt.close(fig)
106
107 def plot_through_thickness_y(z_vals, sig_x, sig_y, sig_xy, p, bc):
108     fig = plt.figure(figsize=(10,8))
109     plt.plot(sig_y, z_vals, linewidth=2)
110     plt.grid(True)
111     plt.xlabel(r'$\sigma_y$')
112     plt.ylabel('z (m)')
113     plt.title(r'$\sigma_y$ vs z')
114     plt.suptitle(f'Through-thickness stresses at center, Boundary Condition: {bc_label(bc)}', p='{p}')
115     filename = os.path.join(output_folder,
116                             f"through_thickness_y_p{p}_{bc}.png")
117     plt.savefig(filename, dpi=300, bbox_inches='tight')
118     plt.close(fig)
119
120 def plot_through_thickness_xy(z_vals, sig_x, sig_y, sig_xy, p, bc):
121     fig = plt.figure(figsize=(10,8))
122     plt.plot(sig_xy, z_vals, linewidth=2)
123     plt.grid(True)
124     plt.xlabel(r'$\sigma_{xy}$')
125     plt.ylabel('z (m)')
126     plt.title(r'$\sigma_{xy}$ vs z')
127     plt.suptitle(f'Through-thickness stresses at center, Boundary Condition: {bc_label(bc)}', p='{p}')
128     filename = os.path.join(output_folder,
129                             f"through_thickness_xy_p{p}_{bc}.png")
130     plt.savefig(filename, dpi=300, bbox_inches='tight')
131     plt.close(fig)
132
133 def assemble_plate_system(p, a, b, q, D, nu, bc_type):
134     n_nodes = p + 1
135     total_dof = n_nodes * n_nodes
136     K = np.zeros((total_dof, total_dof))
137     F = np.zeros(total_dof)
138
139     gauss_points_x, gauss_weights_x = gauss_quadrature(10, 0, a)
140     gauss_points_y, gauss_weights_y = gauss_quadrature(10, 0, b)
141
142     for ix, (x, wx) in enumerate(zip(gauss_points_x, gauss_weights_x)):
143         phi_x = np.zeros(n_nodes)
144         d2phi_x = np.zeros(n_nodes)
145         for i in range(1, n_nodes+1):
146             val, _, d2val = hermite_shape(p, i, x, a)
147             phi_x[i-1] = val
148             d2phi_x[i-1] = d2val
149
150         for iy, (y, wy) in enumerate(zip(gauss_points_y, gauss_weights_y)):
151             weight = wx * wy
152             phi_y = np.zeros(n_nodes)
153             d2phi_y = np.zeros(n_nodes)
154             for j in range(1, n_nodes+1):
155                 val, _, d2val = hermite_shape(p, j, y, b)
156                 phi_y[j-1] = val
157                 d2phi_y[j-1] = d2val
158
159                 for m in range(1, n_nodes+1):
160                     for n in range(1, n_nodes+1):
161                         idx = (i-1)*n_nodes + (j-1)
162                         F[idx] += q * (phi_x[i-1] * phi_y[j-1]) * weight
163                         phi_xx = d2phi_x[i-1] * phi_y[j-1]
164                         phi_yy = phi_x[i-1] * d2phi_y[j-1]
165                         phi_xy = first_derivative(p, i, x, a) *
166                             first_derivative(p, j, y, b)
167                         integrand = (phi_xx * phi_xx_m + phi_yy *
168                                     2*(1-nu)*phi_xy * phi_yy_m +
169                                     nu*(phi_xx * phi_yy_m + phi_xy *
170                                         phi_xx_m))
171                         K[idx, idx] += D * integrand * weight
172
173     constrained = determine_boundary_indices(p, bc_type)
174     for index in constrained:
175         K[index, :] = 0
176         K[:, index] = 0
177         K[index, index] = 1
178         F[index] = 0
179
180     return K, F
181
182 def compute_plate_deflection(p, coeff, a, b):
183     n_points = 21
184     x_coords = np.linspace(0, a, n_points)
185     y_coords = np.linspace(0, b, n_points)
186     disp = np.zeros((n_points, n_points))
187     n_nodes = p + 1
188
189     for i, x in enumerate(x_coords):
190         phi_x = np.array([hermite_shape(p, i+1, x, a)[0] for i in
191                           range(n_nodes)])
192         for j, y in enumerate(y_coords):
193             phi_y = np.array([hermite_shape(p, j+1, y, b)[0] for j in
194                               range(n_nodes)])
195             summ = 0
196             index = 0
197             for ix in range(n_nodes):
198                 for iy in range(n_nodes):
199                     summ += coeff[index] * phi_x[ix] * phi_y[iy]
200                     index += 1
201             disp[j, i] = summ
202     X, Y = np.meshgrid(x_coords, y_coords)
203     return X, Y, disp
204
205 def evaluate_derivatives(p, coeff, x, y, a, b):
206     n_nodes = p + 1
207     val = 0
208     d2x = 0
209     d2y = 0
210     dxy = 0
211
212     phi_x = np.zeros(n_nodes)
213     d2phi_x = np.zeros(n_nodes)
214     for i in range(1, n_nodes+1):
215         v, dv, d2v = hermite_shape(p, i, x, a)
216         phi_x[i-1] = v
217         d2phi_x[i-1] = dv
218         d2phi_x[i-1] = d2v
219
220     phi_y = np.zeros(n_nodes)
221     d2phi_y = np.zeros(n_nodes)
222     for j in range(1, n_nodes+1):
223         v, dv, d2v = hermite_shape(p, j, y, b)
224         phi_y[j-1] = v
225         d2phi_y[j-1] = dv
226         d2phi_y[j-1] = d2v
227
228     index = 0
229     for i in range(n_nodes):
230         for j in range(n_nodes):
231             c = coeff[index]
232             val += c * (phi_x[i] * phi_y[j])
233             d2x += c * (d2phi_x[i] * phi_y[j])
234             d2y += c * (phi_x[i] * d2phi_y[j])
235             dxy += c * (dphi_x[i] * dphi_y[j])
236             index += 1
237
238     return val, d2x, d2y, dxy
239
240 def hermite_shape(p, local_index, X, A):
241     if p == 3:
242         return cubic_shape(local_index, X, A)
243     elif p == 4:
244         if local_index <= 4:
245             return cubic_shape(local_index, X, A)
246         else:
247             return pob2_shape(X, A)
248     elif p == 5:
249         if local_index <= 4:
250             return cubic_shape(local_index, X, A)
251         elif local_index == 5:
252             return pob2_shape(X, A)
253         else:
254             return pob3_shape(X, A)
255
256 def cubic_shape(index, x, A):
257     xi = x / A
258     if index == 1:
259         ref, dref, d2ref = H1(xi)
260         value = ref
261         dvalue = dref / A
262         d2value = d2ref / (A**2)
263     elif index == 2:
264         ref, dref, d2ref = H2(xi)
265         value = A * ref
266         dvalue = dref / A
267         d2value = d2ref / (A**2)
268     elif index == 3:
269         ref, dref, d2ref = H3(xi)
270         value = ref
271         dvalue = dref / A
272         d2value = d2ref / (A**2)
273     elif index == 4:
274         ref, dref, d2ref = H4(xi)
275         value = A * ref
276         dvalue = dref / A
277         d2value = d2ref / A
278     return value, dvalue, d2value
279
280 def H1(xi):
281     v = 1 - 3*xi**2 + 2*xi**3
282     dv = -6*xi + 6*xi**2
283     d2v = -6 + 12*xi
284     return v, dv, d2v
285
286 def H2(xi):
287     v = xi*(1 - 2*xi + xi**2)
288     dv = (1 - 2*xi + xi**2) + xi*(-2 + 2*xi)
289     d2v = -4 + 6*xi
290     return v, dv, d2v
291
292 def H3(xi):
293     v = 3*xi**2 - 2*xi**3
294     dv = 6*xi - 6*xi**2
295     d2v = 6 - 12*xi
296     return v, dv, d2v
297
298 def H4(xi):
299     v = xi**2*(xi - 1)
300     dv = 3*xi**2 - 2*xi
301     d2v = 6*xi - 2
302     return v, dv, d2v
303
304 def pob2_shape(x, A):
305     f1 = x**2; df1 = 2*x; d2f1 = 2
306     f2 = (A - x)**2; df2 = -2*(A - x); d2f2 = 2
307     scale = 1 / (A**4)
308     v = scale * (f1 + f2)
309     dv = scale * (df1 + df2 + f1 * df2)
310     d2v = scale * (d2f1 + f2 + 2*df1*df2 + f1 * d2f2)
311     return v, dv, d2v
312
313 def pob3_shape(x, A):
314     f1 = x**3; df1 = 3*x**2; d2f1 = 6*x
315     f2 = (A - x)**3; df2 = -3*(A - x)**2; d2f2 = 6*(A - x)
316     scale = 1 / (A**6)
317     prod = f1 * f2
318     d2prod = d2f1 * f2 + f2 * d2f1 + df1 * df2 + f1 * d2f2
319     v = scale * prod
320     dv = scale * d2prod
321     d2v = scale * d2prod
322     return v, dv, d2v
323
324 def first_derivative(p, local_index, x, A):
325     _, d_val, _ = hermite_shape(p, local_index, x, A)
326     return d_val
327
328 def determine_boundary_indices(p, bc_type):
329     n_nodes = p + 1
330     indices = []
331     if bc_type == 'SSSS':
332         remove_x = [1, 3]
333         remove_y = [1, 3]
334         for i in range(1, n_nodes+1):
335             for j in range(1, n_nodes+1):
336                 if i in remove_x or j in remove_y:
337                     indices.append((i-1)*n_nodes + (j-1))
338     elif bc_type == 'CFFF':
339         for i in range(1, n_nodes+1):
340             for j in range(3, n_nodes+1):
341                 indices.append((i-1)*n_nodes + (j-1))
342     return sorted(set(indices))
343
344 def gauss_quadrature(n, a, b):
345     if n == 1:
346         points = np.array([0])
347         weights = np.array([2])
348     elif n == 2:
349         points = np.array([-1/np.sqrt(3), 1/np.sqrt(3)])
350         weights = np.array([1, 1])
351     elif n == 3:
352         points = np.array([-np.sqrt(3/5), 0, np.sqrt(3/5)])
353         weights = np.array([0.55555, 0.88889, 0.55555])
354     elif n == 4:
355         points = np.array([-0.86111, -0.33990, 0.33990, 0.86111])
356         weights = np.array([0.34780, 0.65210, 0.65210, 0.34780])
357     elif n == 5:
358         points = np.array([-0.90610, -0.53840, 0, 0.53840, 0.90610])
359         weights = np.array([0.23690, 0.47860, 0.56880, 0.47860, 0.23690])
360     elif n == 6:
361         points = np.array([-0.93247, -0.66121, -0.23862,
362                             0.23862, 0.66121, 0.93247])
363         weights = np.array([0.17132, 0.36076, 0.46791, 0.46791, 0.36076,
364                             0.17132])
365     elif n == 7:
366         points = np.array([-0.94911, -0.74153, -0.40585,
367                             0, 0.40585, 0.74153, 0.94911])
368         weights = np.array([0.12948, 0.27971, 0.38183,
369                             0.47196, 0.38183, 0.27971, 0.12948])
370     elif n == 8:
371         points = np.array([-0.96029, -0.79667, -0.52553, -0.18343,
372                             0.18343, 0.52553, 0.79667, 0.96029])
373         weights = np.array([0.10123, 0.22238, 0.31371, 0.36268,
374                             0.36268, 0.31371, 0.22238, 0.10123])
375     elif n == 9:
376         points = np.array([-0.96816, -0.83603, -0.61337,
377                             -0.32425, 0, 0.32425, 0.61337, 0.83603,
378                             0.96816])
379         weights = np.array([0.08127, 0.18065, 0.26061,
380                             0.31235, 0.33024, 0.31235, 0.26061, 0.18065,
381                             0.08127])
382     elif n == 10:
383         points = np.array([-0.97391, -0.86506, -0.67941, -0.43340,
384                             -0.14887, 0.14887, 0.43340, 0.67941,
385                             0.86506, 0.97391])
386         weights = np.array([0.06667, 0.14945, 0.21909, 0.26927,
387                             0.29552, 0.29552, 0.26927, 0.21909,
388                             0.14945, 0.06667])
389
390     mid = 0.5*(a+b)
391     half_length = 0.5*(b-a)
392     transformed_points = mid + half_length * points
393     transformed_weights = half_length * weights
394     return transformed_points, transformed_weights
395
396 def main():
397     E = 200e9
398     nu = 0.3
399     h = 4e-3
400     a = 0.5
401     b = 0.5
402     qo = 1000
403     D = E * h**3 / (12*(1 - nu**2))
404
405     p_values = [3, 4, 5]
406     boundary_conditions = ['SSSS', 'CFFF']
407     yield_stress = 450e6
408
409     for p in p_values:
410         for bc in boundary_conditions:
411             stiffness, force = assemble_plate_system(p, a, b, qo, D, nu, bc)
412             coeff = np.linalg.solve(stiffness, force)
413
414             X, Y, displacement = compute_plate_deflection(p, coeff, a, b)
415             plot_surface_disp(X, Y, displacement, p, bc, a, b)
416             plot_contour_disp(X, Y, displacement, p, bc, a, b)
417
418             Nx, Ny = 31, 31
419             x_grid = np.linspace(0, a, Nx)
420             y_grid = np.linspace(0, b, Ny)
421             X_grid, Y_grid = np.meshgrid(x_grid, y_grid)
422             top_z = h/2
423
424             vm_stress = np.zeros((Ny, Nx))
425             max_vm = -np.inf
426             max_xy = (0, 0)
427
428             for ix, x in enumerate(x_grid):
429                 for iy, y in enumerate(y_grid):
430                     _, zxx, zyy, zxy = evaluate_derivatives(p, coeff, x, y,
431                                                             a, b)
432                     sig_x = -(E/(1+nu**2)) * top_z * (zxx + neu*zxy)
433                     sig_y = -(E/(1+nu**2)) * top_z * (zxy + neu*zxx)
434                     sig_xy = (E/(1+nu**2)) * top_z * zxy
435                     vm = np.sqrt(0.5*((sig_x - sig_y)**2 + sig_x**2 +
436                                     sig_y**2) + 3*(sig_xy**2))
437                     vm_stress[iy, ix] = vm
438                     if vm > max_vm:
439                         max_vm = vm
440                         max_xy = (x, y)
441
442             plot_vm_contour(X_grid, Y_grid, vm_stress, p, bc, a, b)
443             q_yield = (yield_stress / max_vm) * qo
444             print(f'p={p}, bc={bc}: Max VM Stress={max_vm:.6e} at
445                   ({max_xy[0]:.2f}, {max_xy[1]:.2f})')
446             print(f'p={p}, bc={bc}: q_yield={q_yield:.2e} N/m²')
447
448             strain_energy = 0.5 * coeff.T @ stiffness @ coeff
449             print(f'p={p}, bc={bc}: Strain Energy = {strain_energy:.2e}
450                   J\n')
451
452             center_x, center_y = a/2, b/2
453             z_positions = np.linspace(-h/2, h/2, 51)
454             _, zxx_c, zyy_c, zxy_c = evaluate_derivatives(p, coeff, center_x,
455                                                             center_y, a, b)
456             sig_x_profile = np.array([-E/(1+nu**2)) * z * (zxx_c +
457                                                             neu*zyy_c) for z in z_positions])
458             sig_y_profile = np.array([-E/(1+nu**2)) * z * (zyy_c +
459                                                             neu*zxx_c) for z in z_positions])
460             sig_xy_profile = np.array([-E/(1+nu**2)) * z * zxy_c for z in
461                                       z_positions])
462
463             plot_through_thickness_x(z_positions, sig_x_profile,
464                                     sig_y_profile, sig_xy_profile, p, bc)
465             plot_through_thickness_y(z_positions, sig_x_profile,
466                                     sig_y_profile, sig_xy_profile, p, bc)
467             plot_through_thickness_xy(z_positions, sig_x_profile,
468                                     sig_y_profile, sig_xy_profile, p, bc)
469
470             plot_cubic_hermite_functions()
471
472 if __name__ == "__main__":
473     main()
474     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
```