

source_code.py

```
001| ''' wheel spacing: 52 176 164 176 164 176 52
002| 0 < x_train < 240 mm
003| force exerted by each wheel = 400/6 N
004| '''
005| import numpy as np
006| import matplotlib.pyplot as plt
007| import math
008|
009|
010| L = 1200
011|
012|
013| def create_x_bridge():
014|     ''' Return a list x_bridge of length L'''
015|     global L
016|     x_bridge = []
017|
018|     for i in range(L + 2):
019|         x_bridge.append(i)
020|
021|
022|     return x_bridge
023|
024|
025|
026| def loc_of_wheels(x_train):
027|     ''' Return the location of wheels given x_train'''
028|     w1 = 52 + x_train
029|     w2 = 52 + 176 + x_train
030|     w3 = 52 + 176 + 164 + x_train
031|     w4 = 52 + 176 + 164 + 176 + x_train
032|     w5 = 52 + 176 + 164 + 176 + 164 + x_train
033|     w6 = 52 + 176 + 164 + 176 + 164 + 176 + x_train
034|
035|
036|     return w1, w2, w3, w4, w5, w6
037|
038| # print(loc_of_wheels(164))
039| # L = 1200
040| # w1, w2, w3, w4, w5, w6 = loc_of_wheels(164)
041| def reaction_forces(w1, w2, w3, w4, w5, w6):
042|     global L
043|     By = (400/ 6 * (w1 + w2 + w3 + w4 + w5 + w6)) / L
044|     Ay = 400 - By
045|
```

```

046 |
047 |     return Ay, By
048 |
049 | # print(reaction_forces(w1, w2, w3, w4, w5, w6))
050 |
051 |
052 |
053 | def create_V_M_lists(x_train):
054 |     ''' Return the shear force and bending moment along the
beam when
055 |     the train is at x_train
056 |     V in N
057 |     M in Nmm
058 |     '''
059 |
060 |     global L
061 |     w1, w2, w3, w4, w5, w6 = loc_of_wheels(x_train)
062 |     Ay, By = reaction_forces(w1, w2, w3, w4, w5, w6)
063 |     x_bridge = create_x_bridge()
064 |
065 |     V = [0]
066 |
067 |     for i in range(1, w1 + 1):
068 |         V.append(Ay)
069 |
070 |     for i in range(w1 + 1, w2 + 1):
071 |         V.append(Ay - 400 / 6)
072 |
073 |     for i in range(w2 + 1, w3 + 1):
074 |         V.append(Ay - 400 / 6 * 2)
075 |
076 |     for i in range(w3 + 1, w4 + 1):
077 |         V.append(Ay - 400 / 6 * 3)
078 |
079 |     for i in range(w4 + 1, w5 + 1):
080 |         V.append(Ay - 400 / 6 * 4)
081 |
082 |     for i in range(w5 + 1, w6 + 1):
083 |         V.append(Ay - 400 / 6 * 5)
084 |
085 |     for i in range(w6 + 1, L + 1):
086 |         V.append(-By)
087 |
088 |     V.append(0)
089 |
090 |     M = [0]
091 |

```

```

092 |     for i in range(1, w1 + 1):
093 |         M.append(Ay * i)
094 |
095 |     for i in range(w2 - w1):
096 |         M.append(Ay * w1 + V[i + w1] * i)
097 |
098 |     for i in range(w3 - w2):
099 |         M.append(Ay * w1 + V[w2] * (w2 - w1) + V[i + w2] * i)
100 |
101 |     for i in range(w4 - w3):
102 |         M.append(Ay * w1 + V[w2] * (w2 - w1) + V[w3] * (w3 -
w2) + V[i + w3] * i)
103 |
104 |     for i in range(w5 - w4):
105 |         M.append(Ay * w1 + V[w2] * (w2 - w1) + V[w3] * (w3 -
w2) + V[w4] * (w4 - w3) + \
106 |             V[i + w4] * i)
107 |
108 |     for i in range(w6 - w5):
109 |         M.append(Ay * w1 + V[w2] * (w2 - w1) + V[w3] * (w3 -
w2) + V[w4] * (w4 - w3) + \
110 |             V[w5] * (w5 - w4) + V[i + w5] * i)
111 |
112 |     for i in range(L - w6):
113 |         M.append(Ay * w1 + V[w2] * (w2 - w1) + V[w3] * (w3 -
w2) + V[w4] * (w4 - w3) + \
114 |             V[w5] * (w5 - w4) + V[w6] * (w6 - w5) - By * i)
115 |
116 |     M.append(0)
117 |
118 |
119 |     return V, M
120 |
121 |
122 |
123 |
124 |
125 |
126 | def graph_SFD_BMD(x_train):
127 |
128 |     '''Take in the position of the train, x_train and graphs
the SFD and BMD'''
129 |
130 |     w1, w2, w3, w4, w5, w6 = loc_of_wheels(x_train)
131 |     Ay, By = reaction_forces(w1, w2, w3, w4, w5, w6)
132 |     x_bridge = create_x_bridge()
133 |     x_axis = [0] * (L + 2)

```

```

134 | V, M = create_V_M_lists(x_train)
135 |
136 | plt.figure(f"SFD (x = {x_train})")
137 | plt.plot(x_bridge, x_axis)
138 | plt.title("Shear Force Diagram")
139 | plt.ylabel("Shear Force (N)")
140 | plt.xlabel("Distance Along Bridge (mm)")
141 |
142 | # prints the shear force at each location
143 | print("x = 0, V = 0")
144 | print(f"x = {w1}, V = {Ay}")
145 | print(f"x = {w2}, V = {Ay - 400 / 6}")
146 | print(f"x = {w2}, V = {Ay - 400 / 6 * 2}")
147 | print(f"x = {w3}, V = {Ay - 400 / 6 * 3}")
148 | print(f"x = {w4}, V = {Ay - 400 / 6 * 4}")
149 | print(f"x = {w5}, V = {Ay - 400 / 6 * 5}")
150 | print(f"x = {w6}, V = {-By}")
151 | print(f"x = {L}, V = 0")
152 |
153 | plt.plot(x_bridge, V)
154 | plt.show()
155 |
156 | plt.figure(f"BMD (x = {x_train})")
157 | plt.title("Bending Moment Diagram")
158 | plt.xlabel("Distance Along Bridge (mm)")
159 | plt.ylabel("Bending Moment (Nmm)")
160 | plt.plot(x_bridge, x_axis)
161 |
162 | # graphs the bending moment at each location
163 | print("x = 0, M = 0")
164 | print(f"x = {w1}, M = {Ay * w1}")
165 | print(f"x = {w2}, M = {Ay * w1 + V[w2] * (w2 - w1)}")
166 | print(f"x = {w3}, M = {Ay * w1 + V[w2] * (w2 - w1) +
V[w3] * (w3 - w2)}")
167 | print(f"x = {w4}, M = {Ay * w1 + V[w2] * (w2 - w1) +
V[w3] * (w3 - w2) + V[w4] * (w4 - w3)}")
168 | print(f"x = {w5}, M = {Ay * w1 + V[w2] * (w2 - w1) +
V[w3] * (w3 - w2) + V[w4] * (w4 - w3) + V[w5] * (w5 - w4)}")
169 | print(f"x = {w6}, M = {Ay * w1 + V[w2] * (w2 - w1) +
V[w3] * (w3 - w2) + V[w4] * (w4 - w3) + V[w5] * (w5 - w4) + V[w6]
* (w6 - w5)}")
170 | print(f"x = {L}, M = 0")
171 |
172 | plt.plot(x_bridge, M)
173 | plt.show()
174 |
175 |

```

```

176|
177| def create_V_M_envelopes():
178|     ''' Iterate through lists V, M and find the maximum V and
M at
179|     every location on the bridge
180|
181|     Return 2 dictionaries:
182|     all_V_at_every_loc
183|     and
184|     all_M_at_every_loc
185|
186|     which store the positions on the bridge as keys and the
list of
187|     possible V's and M's as values
188|
189|     '''
190|     global L
191|
192|     all_V_at_every_loc = dict()
193|     all_M_at_every_loc = dict()
194|     # all_V_at_every_loc[x_bridge] = list of all V's in all
240 loading positions
195|     for x_bridge in range(L + 2):
196|         all_V_at_every_loc[x_bridge] = []
197|         all_M_at_every_loc[x_bridge] = []
198|
199|     max_V = 0
200|     max_M = 0
201|     max_V_bridge = 0
202|     max_M_bridge = 0
203|     max_V_train = 0
204|     max_M_train = 0
205|
206|     for x_train in range(241):
207|         V, M = create_V_M_lists(x_train)
208|
209|         for x_bridge in range(L + 2):
210|             # list of V and M values at location x_bridge
211|             # iterate through x_bridge and find the force at
x_train, x_bridge
212|
213|         all_V_at_every_loc[x_bridge].append(abs(V[x_bridge]))
214|
215|         all_M_at_every_loc[x_bridge].append(abs(M[x_bridge]))
216|
217|         if V[x_bridge] > max_V:
218|             max_V = V[x_bridge]

```

```

217|         max_V_bridge = x_bridge
218|         max_V_train = x_train
219|     if M[x_bridge] > max_M:
220|         max_M = M[x_bridge]
221|         max_M_bridge = x_bridge
222|         max_M_train = x_train
223|
224|
225|
226|     # put the max V and M values at every location to a new
list
227|     # list index = position on bridge
228|
229|     V_envelope = []
230|     M_envelope = []
231|     for x_bridge in range(L + 2):
232|         V_envelope.append(max(all_V_at_every_loc[x_bridge]))
233|         M_envelope.append(max(all_M_at_every_loc[x_bridge]))
234|
235|
236|     return V_envelope, M_envelope
237|
238|
239|
240|
241|
242|
243| def graph_V_M_envelopes():
244|     '''Graphs the shear force and bending moment envelopes'''
245|
246|     global L
247|
248|     x_bridge = create_x_bridge()
249|     V_envelope, M_envelope = create_V_M_envelopes()
250|
251|
252|     plt.figure("Shear Force Envelope")
253|     plt.title("Shear Force Envelope")
254|     plt.ylabel("Maximum Shear Force During Loading (N)")
255|     plt.xlabel("Distance Along Bridge (mm)")
256|     x_axis = [0] * (L + 2)
257|     plt.plot(x_bridge, x_axis)
258|     plt.plot(x_bridge, V_envelope)
259|     plt.show()
260|
261|     print(f"x = 200, V = {V_envelope[201]}")
262|     print(f"x = 300, V = {V_envelope[301]}")

```

```

263| print(f"x = 400, V = {V_envelope[401]}")
264| print(f"x = 600, V = {V_envelope[601]}")
265| print(f"x = 800, V = {V_envelope[801]}")
266| print(f"x = 1000, V = {V_envelope[1001]}")
267|
268| max_V = V_envelope[0]
269| max_V_loc = 0
270| for loc in range(len(V_envelope)):
271|     if V_envelope[loc] > max_V:
272|         max_V = V_envelope[loc]
273|         max_V_loc = loc
274|
275| print(f"Max V = {max_V}, location = {max_V_loc}")
276|
277|
278|
279| plt.figure("Bending Moment Envelope")
280| plt.title("Bending Moment Envelope")
281| plt.ylabel("Maximum Bending Moment During Loading (N)")
282| plt.xlabel("Distance Along Bridge (mm)")
283| plt.plot(x_bridge, M_envelope)
284| plt.show()
285|
286| print(f"x = 200, M = {M_envelope[201]}")
287| print(f"x = 300, M = {M_envelope[301]}")
288| print(f"x = 400, M = {M_envelope[401]}")
289| print(f"x = 600, M = {M_envelope[601]}")
290| print(f"x = 800, M = {M_envelope[801]}")
291| print(f"x = 1000, M = {M_envelope[1001]}")
292|
293| max_M = M_envelope[0]
294| max_M_loc = 0
295| for loc in range(301):
296|     if M_envelope[loc] > max_M:
297|         max_M = M_envelope[loc]
298|         max_M_loc = loc
299|
300| print(f"Max M = {max_M} at 301, location = {max_M_loc}")
301|
302|
303|
304|
305| def centroidal(dim):
306|     '''Take in a nested list dim = [[w1, h1, yb1], [w2, h2,
yb2], ..., [wn, hn, ybn]]
307|     Return the centroidal axis of the shape, yb, in mm
308|     Assume sections are rectangular

```

```

309|
310|     '''
311|     #  $y_b = \sum(w_i * h_i * y_{bi}) / \sum(w_i * h_i)$ 
312|     denominator = 0
313|     numerator = 0
314|     for i in range(len(dim)):
315|         denominator += (dim[i][0] * dim[i][1] * dim[i][2])
316|         numerator += (dim[i][0] * dim[i][1])
317|     return denominator / numerator
318|
319|
320|
321| def I_calculator(dim, yb):
322|     ''' Take in a nested list dim = [[w1, h1, yb1], [w2, h2,
yb2], ..., [wn, hn, ybn]]
323|     and y-bar, yb
324|     Return the second moment of area, I, in mm^4
325|     Assume sections are rectangular
326|
327|     '''
328|     #  $I = \sum(I_{oi} + d^2A)$ 
329|     I = 0
330|     for i in range(len(dim)):
331|         I += ((dim[i][0] * dim[i][1] ** 3) / 12 + ((dim[i][2]
- yb) ** 2) * dim[i][0] * dim[i][1]))
332|
333|     return I
334|
335|
336|
337|
338|
339| # 8 FAILURE CHECKS
340|
341| def tens_comp_demand(M, y_top, yb, I):
342|     # check compression on top of beam
343|     comp = M * (y_top - yb) / I
344|     tens = M * yb / I
345|
346|     return comp, tens
347|
348| def shear_demand(V, Q, I, b):
349|     # check shear
350|     return V * Q / I / b
351|
352|
353| def thin_plate_mid(b, t):

```



```

354|     # mid flange
355|     sigma_crit = 4*(math.pi**2)*(4000*10**6)/12/(1-0.2**2)*(t
/ b)**2
356|
357|     return sigma_crit
358|
359| def thin_plate_side(b, t):
360|     # side flange
361|     sigma_crit = 0.425 * (math.pi**2)*(4000*10**6)/12/
(1-0.2**2)*(t / b)**2
362|     return sigma_crit
363|
364| def thin_plate_web(b, t):
365|     # web members
366|     sigma_crit = 6 * (math.pi**2)*(4000*10**6)/12/
(1-0.2**2)*(t / b)**2
367|
368|     return sigma_crit
369|
370|
371|
372|
373| def shear_thin_plate(h, w, a):
374|     # shear thin plate buckling
375|     sigma_crit = 5 * (math.pi**2)*(4000*10**6)/12/
(1-0.2**2)*((w / h)**2 + (w / a)**2)
376|
377|     return sigma_crit
378|
379|
380|
381| # DEMAND
382|
383| def create_tens_envelope(y_tot, yb, I):
384|     '''Take in the total height of the cross-section, y_tot,
385|     and the height of the centroidal axis, yb
386|
387|     Return a list that has the maximum tensile force at each
388|     bridge location
389|
390|     '''
391|     y_tot *= 10 ** -3
392|     yb *= 10 ** -3
393|     I *= 10 ** -12
394|     M_envelope = create_V_M_envelopes()[1]
395|     tens_demand = []
396|     for M in M_envelope:

```

```

397|         tens_demand.append(tens_comp_demand(M*10**-3, y_tot,
yb, I)[1]\
398|         *10**-6)
399|     return tens_demand
400|
401| def create_comp_envelope(y_tot, yb, I):
402|     '''Take in the total height of the cross-section,
403|     y_tot, the height of the centroidal axis, yb, and
404|     the second moment of area, I in the zone of interest
405|
406|     Return a list that has the maximum compressive force
407|     at each bridge location
408|
409|     '''
410|     y_tot *= 10 ** -3
411|     yb *= 10 ** -3
412|     I *= 10 ** -12
413|     M_envelope = create_V_M_envelopes()[1]
414|     comp_demand = []
415|     for M in M_envelope:
416|         comp_demand.append(tens_comp_demand(M*10**-3, y_tot,\
417|         yb, I)[0]*10**-6)
418|     return comp_demand
419|
420| def create_shear_envelope(Q, I, b):
421|     ''' Take in Q, I and b
422|
423|     Return a list of the maximum shear force at each location
424|     of the bridge
425|
426|     '''
427|
428|
429|     Q *= 10**-9
430|     I *= 10 ** -12
431|     b *= 10 ** -3
432|     x_bridge = create_x_bridge()
433|     V_envelope = create_V_M_envelopes()[0]
434|     shear_stress_demand = []
435|     for V in V_envelope:
436|         shear_stress_demand.append(shear_demand(V, Q, I, b)\
437|         *10**-6)
438|     return shear_stress_demand
439|
440|
441|
442|

```

```

443|
444| def graph_demand_capacity(title, xlabel, ylabel, demand,
capacity, start, end):
445|     ''' Creates a capacity-demand graph'''
446|
447|     plt.title(title)
448|     plt.xlabel(xlabel)
449|     plt.ylabel(ylabel)
450|     x_bridge = create_x_bridge()[start: end + 1]
451|     plt.plot(x_bridge, capacity[start: end + 1])
452|     plt.plot(x_bridge, demand[start: end + 1])
453|     plt.show()
454|
455|
456| # FINAL DESIGN
457|
458|
459| def x_sectional(zone):
460|     global yb, I
461|     yb = centroidal(zone)
462|     # print(f"yb = {round(yb, 3)}")
463|     I = I_calculator(zone, yb)
464|     # print(f"I = {round(I, 3)}")
465|     return yb, I
466|
467|
468| # demand
469| def calculate_demand(y_top, yb, I, Q_glue, Q_cent, start,
end):
470|     '''Calculates all the stresses demanded'''
471|
472|     global comp_stress_envelope, tens_stress_envelope,
glue_shear_stress_envelope,\
473|     matboard_shear_stress_envelope, max_comp_stress,
max_tens_stress,\
474|     matboard_max_shear_stress, glue_max_shear_stress
475|
476|     comp_stress_envelope = create_comp_envelope(y_top, yb, I)
477|     tens_stress_envelope = create_tens_envelope(y_top, yb, I)
478|     glue_shear_stress_envelope =
create_shear_envelope(Q_glue, I, 12.54)
479|     matboard_shear_stress_envelope =
create_shear_envelope(Q_cent, I, 2.54)
480|
481|     max_comp_stress = max(comp_stress_envelope[start: end +
1])
482|     max_tens_stress = max(tens_stress_envelope[start: end +

```

```

1])
483|     matboard_max_shear_stress =
max(matboard_shear_stress_envelope[start: end + 1])
484|     glue_max_shear_stress =
max(glue_shear_stress_envelope[start: end + 1])
485|
486|
487| # capacity
488| def calculate_capacity(b_mid_flange, t_mid_flange,
b_side_flange, t_side_flange, b_web, t_web, h_diaphragm,
a_diaphragm):
489|     '''Calculate all the capacities demanded'''
490|
491|     global comp_capacity, tens_capacity, shear_glue_capacity,
\
492|     shear_matboard_capacity, mid_flange_capacity,
side_flange_capacity,\
493|     web_capacity, shear_thin_plate_capacity
494|
495|     comp_capacity = [6] * 1201
496|
497|     tens_capacity = [30] * 1201
498|
499|     shear_glue_capacity = [2] * 1201
500|
501|     shear_matboard_capacity = [4] * 1201
502|
503|     mid_flange_capacity = [thin_plate_mid(b_mid_flange,
t_mid_flange)\
504|         *10**-6] * 1201
505|
506|     side_flange_capacity = [thin_plate_side(b_side_flange,
t_side_flange)\
507|         *10**-6] * 1201
508|
509|     web_capacity = [thin_plate_web(b_web, t_web)*10**-6] *
1201
510|
511|     shear_thin_plate_capacity =
[shear_thin_plate(h_diaphragm, 1.27, \
512|         a_diaphragm)*10**-6] * 1201
513|
514|
515|
516| def graph_everything(start, end, zone):
517|     ''' Graphs all demands and capacities'''
518|

```

```

519|     graph_demand_capacity("Compressive Stress vs Capacity",
"Distance Along Bridge (mm)",\
520|     "Compressive Stress and Capacity", comp_stress_envelope,
comp_capacity, start, end)
521|     print("compressive stress capacity:", comp_capacity[0])
522|     print("maximum compressive stress:", max_comp_stress)
523|     print(f"minimum FOS against compression failure: {6 /
max_comp_stress}")
524|
525|     graph_demand_capacity("Tensile Stress vs Capacity",
"Distance Along Bridge (mm)",\
526|     "Tensile Stress and Capacity", tens_stress_envelope,
tens_capacity, start, end)
527|     print("\ntensile stress capacity:", tens_capacity[0])
528|     print("maximum tensile stress:", max_tens_stress)
529|     print(f"minimum FOS against tension failure: {30 /
max_tens_stress}")
530|
531|     graph_demand_capacity("Glue Shear Demand vs Capacity",
"Distance Along Bridge (mm)",\
532|     "Shear Stress and Capacity", glue_shear_stress_envelope,
shear_glue_capacity, start, end)
533|     print("\nglue shear stress capacity:",
shear_glue_capacity[0])
534|     print("maximum shear stress at glue location:",
glue_max_shear_stress)
535|     print(f"minimum FOS against glue shear failure: {2 /
glue_max_shear_stress}")
536|
537|     graph_demand_capacity("Matboard Shear Demand vs
Capacity", "Distance Along Bridge (mm)",\
538|     "Shear Stress and Capacity",
matboard_shear_stress_envelope, shear_matboard_capacity, start,
end)
539|     print("\nmatboard shear stress capacity: 4")
540|     print("maximum shear stress in matboard:",
matboard_max_shear_stress)
541|     print(f"minimum FOS against matboard shear failure: {4 /
matboard_max_shear_stress}")
542|
543|     graph_demand_capacity("Mid-Flange Buckling Stress vs
Capacity", "Distance Along Bridge(mm)",\
544|     "Buckling Stress and Capacity", comp_stress_envelope,
mid_flange_capacity, start, end)
545|     print("\nmid-flange buckling capacity:",
mid_flange_capacity[0])
546|     print("maximum shear stress in the mid-flange:",

```

```

matboard_max_shear_stress)
547|     print(f"minimum FOS against mid flange buckling:
{mid_flange_capacity[0] / max_comp_stress}")
548|
549|     graph_demand_capacity("Side-Flange Buckling Stress vs
Capacity", "Distance Along Bridge (mm)",\
550|         "Buckling Stress and Capacity", comp_stress_envelope,
side_flange_capacity, start, end)
551|     print("\nside-flange buckling capacity:",
side_flange_capacity[0])
552|     print("maximum shear stress in the mid-flange:",
matboard_max_shear_stress)
553|     print(f"minimum FOS against side flange buckling:
{side_flange_capacity[0] / max_comp_stress}")
554|
555|     graph_demand_capacity("Web Buckling Stress vs Capacity",
"Distance Along Bridge (mm)",\
556|         "Buckling Stress and Capacity", comp_stress_envelope,
web_capacity, start, end)
557|     print("\nweb buckling capacity:", web_capacity[0])
558|     print("maximum shear stress in the mid-flange:",
matboard_max_shear_stress)
559|     print(f"minimum FOS against web buckling:
{web_capacity[0] / max_comp_stress}")
560|
561|     graph_demand_capacity("Shear Buckling Demand vs
Capacity", "Distance Along Bridge (mm)",\
562|         "Shear Buckling Stress and Capacity",
matboard_shear_stress_envelope, shear_thin_plate_capacity, start,
end)
563|     print("\nshear buckling capacity:",
shear_thin_plate_capacity[0])
564|     print("maximum shear stress:", matboard_max_shear_stress)
565|     print(f"minimum FOS against shear buckling:
{shear_thin_plate_capacity[0] / matboard_max_shear_stress}")
566|
567| # design 0
568| design0 = [100, 1.27, 75.635], [2.54, 75-1.27, 38.135], [10,
1.27, 74.365],\
569| [80, 1.27, 0.635]
570|
571| # design iterations
572| design1 = [[100, 2.54, 76.905], [2.54, 75, 37.5], [10, 1.27,
73.73], [71.46,\
573| 1.27, 0.635]]
574|
575| design2 = [[200, 2.54, 76.27], [2.54, 75, 37.5], [20, 1.27,

```

```

74.365], [2.54, 1.27, 38.135],\
576| [77.46, 1.27, 0.635]]
577|
578| design3 = [[100, 2.54, 82.54], [2.54, 90, 40.635],
[55-2*1.27, 1.27, 1.27/2+80],\
579| [1.27, 80-1.27, (80+1.27)/2], [55, 1.27, 0.635]]
580|
581| # final design
582| A = [[100, 2.54, 82.54], [75, 1.27, 80.635], [20, 1.27,
79.365],\
583| [2.54, 80, 40], [72.46, 1.27, 0.635]]
584|
585| B = [[100, 2.54, 82.54], [2*11.27, 2.54, 80], [2*1.27, 78.73,
39.365],\
586| [72.46, 1.27, 0.635]]
587|
588|
589| design = design0
590| zone = A
591|
592| if design == design0:
593|     Q_glue = 4344.035
594|     Q_cent = 6193.079
595|     x_sectional(design)
596|     calculate_demand(76.27, yb, I, Q_glue, Q_cent, 0, 1200)
597|     calculate_capacity(78.73, 1.27, 10.635, 1.27, 75 - 1.27/2
- yb, 1.27, 73.73, 400 )
598|     graph_everything(0, 1200, design0)
599|
600| elif design == design1:
601|     Q_glue = 8402.955
602|     Q_cent = 9154.180828
603|     x_sectional(design1)
604|     calculate_demand(75+1.27*3, yb, I, Q_glue, Q_cent, 400,
800)
605|     calculate_capacity(80-1.27, 3.81, 10.635, 3.81, 75+3*1.27
- yb, 1.27, 75-1.27, 400)
606|     graph_everything(0, 1200, design1)
607|
608| elif design == design2:
609|     x_sectional(design2)
610|     Q_glue = 4614.672
611|     Q_cent = 14227.9
612|     x_sectional(design2)
613|     calculate_demand(75+1.27*2, yb, I, Q_glue, Q_cent, 0,
1200)
614|     calculate_capacity(80-1.27*2, 2.54, 11.27, 2.54, 76.27 -

```

```

yb, 2.54, 75-1.27, 400)
615|     graph_everything(0, 1200, design2)
616|
617| elif design == design3:
618|     x_sectional(design3)
619|     Q_glue = 12561
620|     Q_cent = 9632.232254
621|     x_sectional(design3)
622|     calculate_demand(83.81, yb, I, Q_glue, Q_cent, 0, 1200)
623|     calculate_capacity(55-1.27, 3.81, 23.135, 2.54, 81.27 -
yb, 1.27, 80-1.27/2, 400)
624|     graph_everything(0, 1200, design3)
625|
626| elif design == final:
627|     if zone == A:
628|         Q_glue = 6883.111
629|         Q_cent = 9549.54895
630|         x_sectional(A)
631|         calculate_demand(83.81, yb, I, Q_glue, Q_cent, 400,
800)
632|         calculate_capacity(73.73, 3.81, 13.77, 2.54, 80.635 -
yb, 1.27, 78.73, 360)
633|         graph_everything(400, 800, zone)
634|     else:
635|         Q_glue = 7590.497
636|         Q_cent = 8959.0227
637|         x_sectional(B)
638|         calculate_demand(83.81, yb, I, Q_glue, Q_cent, 0,
400)
639|         calculate_capacity(73.73, 2.54, 13.77, 2.54, 80.635 -
yb, 1.27, 78.73, 250)
640|         graph_everything(0, 400, zone)
641|
642|
643| # GLUE FAILURE AT SPLICING LOCATIONS
644| sigma = create_shear_envelope(8959.0227*10**-9,
652622.7973685571*10**-12, 2.54*10**-3)[127]
645| FOS = 2/sigma
646| print(FOS) # 1.7211822386750777

```