
revised-ising-class.py

```
1  # We create a 'master' Ising class, with options to gap the spectrum or use mixed correlator  
   information.  
2  import bootstrap  
3  import matplotlib.pyplot as plt  
4  import time, datetime  
5  import datetime  
6  import numpy as np  
7  from matplotlib.backends.backend_pdf import PdfPages  
8  
9  sig_defaults = np.arange(0.5,0.85,0.05).tolist()  
10 eps_defaults = np.arange(1.0,2.2,0.2).tolist()  
11  
12 class Point(object):  
13     def __init__(self, sig, eps, kmax, lmax, mmax, nmax, allowed, run_time, cpu_time):  
14         self.sig = sig  
15         self.eps = eps  
16         self.kmax = kmax  
17         self.lmax = lmax  
18         self.mmax = mmax  
19         self.nmax = nmax  
20         self.allowed = allowed  
21         self.run_time = run_time  
22         self.cpu_time = cpu_time  
23  
24 class Grid(object):  
25     def __init__(self, kmax, lmax, mmax, nmax, allowed_points, disallowed_points, run_time,  
   cpu_time):  
26         self.kmax = kmax  
27         self.lmax = lmax  
28         self.mmax = mmax  
29         self.nmax = nmax  
30         self.allowed_points = allowed_points  
31         self.disallowed_points = disallowed_points  
32         self.run_time = run_time  
33         self.cpu_time = cpu_time  
34  
35 class Ising(object):  
36     def __init__(self, name, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults  
   ):  
37         self.dim = dim  
38         self.gap = gap  
39         self.sig_values = sig_values  
40         self.eps_values = eps_values  
41         self.grid_table = []  
42         self.name = name  
43  
44     # For a given set of conformal blocks, set by kmax and lmax, generate a grids for a specified  
   range of mmax and nmax.  
45     # If we obtain a grid of entirely disallowed points, fill in the rest of the grids for that  
   kmax and lmax.  
46     def iterate_parameters(self, kmax_range, lmax_range, mmax_range, nmax_range):  
47         keys = self.generate_keys(kmax_range, lmax_range, mmax_range, nmax_range)  
48  
49         while len(keys) > 0:
```

```

50     # Used keys will store the keys for which there is already a grid in table.
51     used_keys = []
52     #null_keys = []
53
54     for key in keys:
55         if self.get_grid_index(key) != -1:
56             used_keys.append(key)
57             continue
58         print("Trying kmax = " + str(key[0]) + ", lmax = " + str(key[1]) + ", mmax = " + str(key
59             [2]) + ", nmax = " + str(key[3]))
60         self.determine_grid(key)
61         used_keys.append(key)
62
63         # If the grid has only disallowed points...
64         if self.grid_table[self.get_grid_index(key)].allowed_points == []:
65             print ("In the if statement.")
66             k = key[0]
67             l = key[1]
68             m = key[2]
69             n = key[3]
70
71             null_keys = [key for key in keys if key not in used_keys and key[0] == k and key[1] ==
72                 l and key[2] >= m and key[3] >= n]
73
74             for key in null_keys:
75                 if self.get_grid_index(key) != -1:
76                     used_keys.append(key)
77                     continue
78                 #grid = Grid(*key, [], [])
79                 grid = Grid(*(key + [[], [], 0, 0]))
80
81                 for sig in self.sig_values:
82                     for eps in self.eps_values:
83                         grid.disallowed_points.append((sig, eps))
84
85                 self.grid_table.append(grid)
86                 self.save_grid(grid, self.name)
87
88             break
89
90         # We remove all keys from the list that we are done with.
91         keys = [key for key in keys if key not in null_keys and key not in used_keys]
92         null_keys = []
93
94     '''
95     # Saves the data as an executable file that will repopulate the table attribute.
96     # Note, we now do this as we go, instead of at the end, to avoid loss of mass data.
97     def save_to_file(self, name):
98         with open(name + ".py", 'w') as file:
99             file.write("self.table = []\n")
100             for grid in self.table:
101                 file.write("kmax = " + str(grid.kmax) + "\n")
102                 file.write("lmax = " + str(grid.lmax) + "\n")
103                 file.write("mmax = " + str(grid.mmax) + "\n")
104                 file.write("nmax = " + str(grid.nmax) + "\n")
105                 file.write("allowed_points = " + str(grid.allowed_points) + "\n")
106                 file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")

```

```

106         file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
107                 disallowed_points))" + "\n")
108     '''
109     def save_grid(self, grid, name):
110         with open(name + ".py", 'a') as file:
111             file.write("kmax = " + str(grid.kmax) + "\n")
112             file.write("lmax = " + str(grid.lmax) + "\n")
113             file.write("mmax = " + str(grid.mmax) + "\n")
114             file.write("nmax = " + str(grid.nmax) + "\n")
115             file.write("allowed_points = " + str(grid.allowed_points) + "\n")
116             file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")
117             file.write("run_time = " + str(grid.run_time) + "\n")
118             file.write("cpu_time = " + str(grid.cpu_time) + "\n")
119             file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
120                 disallowed_points, run_time, cpu_time))" + "\n")
121
122     # Recoveres a table stored to a file.
123     def recover_grid_table(self, file_name):
124         exec(open(file_name + ".py").read())
125
126     # Searches table of grids for index matching the input key. Returns -1 if not found.
127     def get_grid_index(self, key):
128         for i in range(0, len(self.grid_table)):
129             if self.grid_table[i].kmax == key[0] and self.grid_table[i].lmax == key[1] and self.
130                 grid_table[i].mmax == key[2] and self.grid_table[i].nmax == key[3]:
131                 return i
132         return -1
133
134     # Plots a single grid, specified by a key. Note grid must be in grid_table.
135     def plot_grid(self, key, name):
136         grid = grid_table[self.get_grid_index(key)]
137         allowed_sig = [points[0] for points in grid.allowed_points]
138         allowed_eps = [points[1] for points in grid.allowed_points]
139         disallowed_sig = [points[0] for points in grid.disallowed_points]
140         disallowed_eps = [points[1] for points in grid.disallowed_points]
141
142         # Plot a grid.
143         plt.plot(allowed_sig, allowed_eps, 'r+')
144         plt.plot(disallowed_sig, disallowed_eps, 'b+')
145         plt.title('kmax : ' + grid.kmax.__str__() + " " +
146                 'lmax : ' + grid.lmax.__str__() + " " +
147                 'mmax : ' + grid.mmax.__str__() + " " +
148                 'nmax : ' + grid.nmax.__str__())
149
150     # Plots and saves a series of grids to an output PDF file.
151     # Takes as input parameter values for which we want plotted grids, and the desired PDF file
152     # name.
153     def plot_grids(self, keys, file_name, plots_per_page, grid_size):
154         table = self.generate_table(keys)
155         pdf_pages = PdfPages(file_name + ".pdf")
156
157         # Define the number of plots per page and the size of the grid board.
158         nb_plots = len(grid_table)
159         # nb_plots_per_page = 6
160         nb_pages = int(np.ceil(nb_plots / float(plots_per_page)))
161         # grid_size=(3,2)

```

```

160
161 # This will define which row of the grid we are on.
162 row_index = 0
163
164 # We go through each 'grid' in 'grid_table', generating a plot for each.
165 for i in range(nb_plots):
166     # To begin, declare a new figure / page if we have exceeded limit of the last page.
167     if i % plots_per_page == 0:
168         fig = plt.figure(figsize=(8.27, 11.69), dpi=100)
169
170     # Now, add a plot for the current grid on the grid board.
171     plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
172     if i % grid_size[1] == 1:
173         row_index += 1
174
175     # Handle our data. Retrieve isolated points for plotting from our input grid_table of Grid
176     # objects.
177     allowed_sig = [points[0] for points in table[i].allowed_points]
178     allowed_eps = [points[1] for points in table[i].allowed_points]
179     disallowed_sig = [points[0] for points in table[i].disallowed_points]
180     disallowed_eps = [points[1] for points in table[i].disallowed_points]
181
182     # Plot a grid.
183     plt.plot(allowed_sig, allowed_eps, 'r+')
184     plt.plot(disallowed_sig, disallowed_eps, 'b+')
185     plt.title('kmax : ' + table[i].kmax.__str__() + " " +
186             'lmax : ' + table[i].lmax.__str__() + " " +
187             'mmax : ' + table[i].mmax.__str__() + " " +
188             'nmax : ' + table[i].nmax.__str__())
189
190     # If we have filled a page, or have reached the end of our plots, tight-pack and save the
191     # page.
192     if (i + 1) % plots_per_page == 0 or (i + 1) == nb_plots:
193         plt.tight_layout()
194         pdf_pages.savefig(fig)
195         row_index = 0
196
197 pdf_pages.close()
198
199 # Returns a key or list of keys generated by the input parameter ranges.
200 def generate_keys(self, kmax_range, lmax_range, mmax_range, nmax_range):
201     if type(kmax_range) == int:
202         kmax_range = [kmax_range]
203     if type(lmax_range) == int:
204         lmax_range = [lmax_range]
205     if type(mmax_range) == int:
206         mmax_range = [mmax_range]
207     if type(nmax_range) == int:
208         nmax_range = [nmax_range]
209     keys = []
210     for kmax in kmax_range:
211         for lmax in lmax_range:
212             for mmax in mmax_range:
213                 for nmax in nmax_range:
214                     key = [kmax, lmax, mmax, nmax]
215                     keys.append(key)
216     return keys

```

```

216 # Generates a subtable table of desired, already determined grids from main table.
217 # Gives a warning message if a grid isn't found.
218 def generate_table(self, keys):
219     # table to store the resulting grids.
220     table = []
221     for key in keys:
222         if self.get_grid_index(key) == -1:
223             print("Grid at kmax = " + str(key[0]) + ", " +
224                   "lmax = " + str(key[1]) + ", " +
225                   "mmax = " + str(key[2]) + ", " +
226                   "nmax = " + str(key[3]) + ", " + "does not exist.")
227         else:
228             table.append(self.table[self.get_grid_index(key)])
229
230     return table
231
232 # Takes two keys and returns a dictionary with the direction of every point.
233 def changes(self, key1, key2):
234     changes = {}
235     allowed_one = self.grid_table[self.get_grid_index(key1)].allowed_points
236     allowed_two = self.grid_table[self.get_grid_index(key2)].allowed_points
237
238     for sig in self.sig_values:
239         for eps in self.eps_values:
240             if (sig, eps) in allowed_one and (sig, eps) in allowed_two:
241                 changes[(sig, eps)] = 0
242             if (sig, eps) not in allowed_one and (sig, eps) not in allowed_two:
243                 changes[(sig, eps)] = 0
244             if (sig, eps) in allowed_one and (sig, eps) not in allowed_two:
245                 changes[(sig, eps)] = -1
246             if (sig, eps) not in allowed_one and (sig, eps) in allowed_two:
247                 changes[(sig, eps)] = 1
248     return changes
249
250 # grid_size is a tuple of (rows, columns).
251 def plot_changes(self, keys, file_name, plots_per_page, grid_size):
252     pdf_pages = PdfPages(file_name + ".pdf")
253
254     # Define the number of plots per page and the size of the grid board.
255     # We have one less plots than grids.
256     nb_plots = len(keys)
257     # nb_plots_per_page = 6
258     nb_pages = int(np.ceil(nb_plots / float(plots_per_page)))
259     # grid_size=(3,2)
260
261     # This will define which row of the grid we are on.
262     row_index = 0
263
264     # We go through each 'grid' in 'grid_table', generating a plot for each.
265     for i in range(nb_plots):
266         # To begin, declare a new figure / page if we have exceeded limit of the last page.
267         # 8.27 x 11.69 dimensions of A4 page in inches. DPI - dots per inch (resolution.)
268         if i % plots_per_page == 0:
269             fig = plt.figure(figsize=(8.27, 11.69), dpi=100)
270
271         # Now, add a plot for the current grid on the grid board.
272         plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
273         if i % grid_size[1] == 1:

```

```

274         row_index += 1
275
276     # We want the first grid to compare all changes to.
277     if i == 0:
278         grid = self.grid_table[self.get_grid_index(keys[i])]
279         allowed_sig = [points[0] for points in grid.allowed_points]
280         allowed_eps = [points[1] for points in grid.allowed_points]
281         disallowed_sig = [points[0] for points in grid.disallowed_points]
282         disallowed_eps = [points[1] for points in grid.disallowed_points]
283
284         # Plot the grid.
285         plt.plot(allowed_sig, allowed_eps, 'r+')
286         plt.plot(disallowed_sig, disallowed_eps, 'b+')
287         plt.title('kmax : ' + grid.kmax.__str__() + " " +
288                 'lmax : ' + grid.lmax.__str__() + " " +
289                 'mmax : ' + grid.mmax.__str__() + " " +
290                 'nmax : ' + grid.nmax.__str__())
291
292         y_range = plt.ylim()
293         x_range = plt.xlim()
294
295     else:
296         changes = self.changes(keys[i-1], keys[i])
297         unchanged_points = []
298         to_allowed_points = []
299         to_disallowed_points = []
300         for point in changes:
301             if changes[point] == 0:
302                 unchanged_points.append(point)
303             if changes[point] == 1:
304                 to_allowed_points.append(point)
305             if changes[point] == -1:
306                 to_disallowed_points.append(point)
307
308         unchanged_sig = [points[0] for points in unchanged_points]
309         unchanged_eps = [points[1] for points in unchanged_points]
310         to_disallowed_sig = [points[0] for points in to_disallowed_points]
311         to_disallowed_eps = [points[1] for points in to_disallowed_points]
312         to_allowed_sig = [points[0] for points in to_allowed_points]
313         to_allowed_eps = [points[1] for points in to_allowed_points]
314
315         # Plot a grid.
316         plt.plot(to_allowed_sig, to_allowed_eps, 'r+')
317         plt.plot(to_disallowed_sig, to_disallowed_eps, 'b+')
318         plt.xlim(x_range)
319         plt.ylim(y_range)
320         plt.title('kmax : ' + self.grid_table[self.get_grid_index(keys[i])].kmax.__str__() + " " +
321                 +
322                 'lmax : ' + self.grid_table[self.get_grid_index(keys[i])].lmax.__str__() + " " +
323                 'mmax : ' + self.grid_table[self.get_grid_index(keys[i])].mmax.__str__() + " " +
324                 'nmax : ' + self.grid_table[self.get_grid_index(keys[i])].nmax.__str__())
325
326     # If we have filled a page, or have reached the end of our plots, tight-pack and save the
327     page.
328     if (i + 1) % plots_per_page == 0 or (i + 1) == nb_plots:
329         plt.tight_layout()
330         pdf_pages.savefig(fig)
331         row_index = 0

```

```

330     pdf_pages.close()
331
332
333 class SingleCorrelator(Ising):
334     bootstrap.cutoff=1e-10
335     def __init__(self, name, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults
336         ):
337         self.dim = dim
338         self.gap = gap
339         self.sig_values = sig_values
340         self.eps_values = eps_values
341         self.grid_table = []
342         self.name = name
343
344     # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
345     def determine_grid(self, key):
346         #if self.get_grid_index(key) != -1:
347         start_time=time.time()
348         start_cpu=time.clock()
349         tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
350         tab2 = bootstrap.ConvolvedBlockTable(tab1)
351
352         # Instantiate a Grid object with appropriate input values.
353         # grid=Grid(*key, [], [])
354         grid = Grid(*(key + [[], [], 0, 0]))
355
356         for sig in self.sig_values:
357             for eps in self.eps_values:
358                 sdp = bootstrap.SDP(sig, tab2)
359                 # SDPB will naturally try to parallelize across 4 cores / slots.
360                 # To prevent this, we set its 'maxThreads' option to 1.
361                 # See 'common.py' for the list of SDPB option strings, as well as their default values.
362                 sdp.set_option("maxThreads", 1)
363                 sdp.set_bound(0, float(self.gap))
364                 sdp.add_point(0, eps)
365                 result = sdp.iterate()
366                 if result:
367                     grid.allowed_points.append((sig, eps))
368                 else:
369                     grid.disallowed_points.append((sig, eps))
370
371         # Now append this grid object to the IsingGap grid_table.
372         # Note we will need to implement a look up table to retrieve desired data.
373         end_time=time.time()
374         end_cpu=time.clock()
375         run_time=end_time-start_time
376         cpu_time=end_cpu-start_cpu
377         run_time = datetime.timedelta(seconds = int(end_time - start_time))
378         cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
379
380         grid.run_time = run_time
381         grid.cpu_time = cpu_time
382         self.grid_table.append(grid)
383         self.save_grid(grid, self.name)
384
385     # For mixed correlator, we pass pairs of external scaling dimensions to the SDP.
386     # We copy the content of the triples entering the SDP from the tutorial, same case.
387     # We want to scan over all possible [sig, eps], assuming only one relevant Z2-even and Z2-odd

```

```

operator.
387 # Use a prototype to use the same basis for all SDPs, so we don't need to recalculate bases.
388 # Dump the ConformalBlockTable objects once we have used them to save memory.
389 # Set dualThresholdError to 1e-15.
390 # Use 16 cores for all SDP runs - set maxThreads = 16, speed up the SDP.
391 class MixedCorrelator(Ising):
392     sig_range_def = [0.5179, 0.5186]
393     eps_range_def = [1.4100, 1.4150]
394     sig_step_def = 0.000005
395     eps_step_def = 0.00005
396     bootstrap.cutoff=0
397     def __init__(self, name, dim = 3, sig_range = sig_range_def, eps_range = eps_range_def,
398                 sig_step = sig_step_def, eps_step = eps_step_def):
399         self.name = name
400         self.dim = dim
401         self.sig_range = sig_range
402         self.eps_range = eps_range
403         self.sig_step = sig_step
404         self.eps_step = eps_step
405         self.point_table = []
406         self.grid_table = []
407
408 # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
409 def determine_points(self, key):
410     # key = [self.kmax, self.lmax, self.mmax, self.nmax]
411     #if self.get_grid_index(key) != -1:
412     reference_sdp = None
413
414     # Constant sig-eps lines: eps = sig - c.
415     # Choose a starting point for each line. (0.5179, 1.4110).
416     # g_tab1 and g_tab3 don't change on a given line of constant delta{sig,eps}.
417     sig_values = np.arange(self.sig_range[0], self.sig_range[1] + self.sig_step, self.sig_step).
418         tolist()
419     for eps in np.arange(self.eps_range[0], self.eps_range[1] + self.eps_step, self.eps_step).
420         tolist():
421         # sig_values = []
422         eps_values = []
423         # For each value of x along this line:
424         for sig in sig_values:
425             # sig_values.append(sig)
426             eps_values.append(eps + (sig-self.sig_range[0]))
427
428     # Could initiate all blocks prior to loop here using sig_values[0].
429     # However, want to capture the timing of this within the first point?
430     blocks_initiated = False
431     for i in range(len(sig_values)):
432         sig = sig_values[i]
433         eps = eps_values[i]
434         start_time=time.time()
435         start_cpu=time.clock()
436         # Generate three conformal block tables, two of which depend on the dimension differences
437         .
438         # They need only be calculated once for any given diagonal. They remain constant along
439         this line.
440         # Uses the function above to return the 5 ConvolvedConformalBlocks we need.
441         # The ConvolvedConformalBlock objects inherits the dimension differences from
442         ConformalBlockTable.
443         # We set odd_spins = True for odd those ConvolvedConformalBlocks appearing in odd-sector-

```



```

    odd-spins.
438 # We set symmetric = True where required.
439 if blocks_initiated == False:
440     g_tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
441     g_tab2 = bootstrap.ConformalBlockTable(self.dim, *(key + [eps-sig, sig-eps, "odd_spins
        = True"]))
442     g_tab3 = bootstrap.ConformalBlockTable(self.dim, *(key + [sig-eps, sig-eps, "odd_spins
        = True"]))
443     tab_list = self.convolved_table_list(g_tab1, g_tab2, g_tab3)
444     for tab in [g_tab1, g_tab2, g_tab3]:
445         tab.dump("tab_" + str(tab.delta_12) + "_" + str(tab.delta_34))
446     del tab
447     blocks_initiated = True
448
449 # N.B vec3 & vec2 are 'raw' quads, which will be converted to 1x1 matrices automatically.
450 # Third vector: 0, 0, 1 * table4 with one of each dimension, -1 * table2 with only pair
    [0] dimensions, 1 * table3 with only pair[0] dimensions
451 vec3 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [-1, 2, 0, 0], [1, 3, 0, 0]]
452 # Second vector: 0, 0, 1 * table4 with one of each dimension, 1 * table2 with only pair
    [0] dimensions, -1 * table3 with only pair[0] dimensions
453 vec2 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [1, 2, 0, 0], [-1, 3, 0, 0]]
454 # The first vector has five components as well but they are matrices of quads, not just
    the quads themselves.
455 m1 = [[[1, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
456 m2 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [1, 0, 1, 1]]]
457 m3 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
458 m4 = [[[0, 0, 0, 0], [0.5, 0, 0, 1]], [[0.5, 0, 0, 1], [0, 0, 0, 0]]]
459 m5 = [[[0, 1, 0, 0], [0.5, 1, 0, 1]], [[0.5, 1, 0, 1], [0, 1, 0, 0]]]
460 vec1 = [m1, m2, m3, m4, m5]
461
462 # The first rep must be the singlet even channel, where the unit operator resides.
463 # After this, the order doesn't matter.
464 # Spins for these again go even, even, odd.
465 # The Z2 even sector has only even spins, Z2 odd sector runs over even and odd spins.
466 info = [[vec1, 0, "z2-even-l-even"], [vec2, 0, "z2-odd-l-even"], [vec3, 1, "z2-odd-l-odd"
    ]]
467
468 # We instantiate the SDP object, inputting our vectorial sum info.
469 # dim_list, convolved_block_table_list, vector_types (how they combine to compose sum
    rule).
470 # We use the first calculated SDP object as a prototype for all the rest.
471 # This is because some bounds remain unchanged, no need to recalculate basis.
472 # Basis is independent of external scaling dimensions, cares only of the bounds on
    particular operators.
473 sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
474 if reference_sdp == None:
475     sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
476     reference_sdp = sdp
477 else:
478     sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info, prototype =
        reference_sdp)
479
480 # We assume the continuum in both Z2 odd / even sectors begins at the dimension=3.
481 sdp.set_bound([0, "z2-even-l-even"], self.dim)
482 sdp.set_bound([0, "z2-odd-l-even"], self.dim)
483
484 # Except for the two lowest dimension scalar operators in each sector.
485 sdp.add_point([0, "z2-even-l-even"], eps)

```

```

486         sdp.add_point([0, "z2-odd-l-even"], sig)
487
488         # We expect these calculations to be computationally intensive.
489         # We set maxThreads=16 to parallelise SDPB for all runs.
490         # See 'common.py' for the list of SDPB option strings, as well as their default values.
491         sdp.set_option("maxThreads", 16)
492         sdp.set_option("dualErrorThreshold", 1e-15)
493
494         # Run the SDP to determine if the current operator spectrum is permissible.
495         print("Testing point " + "(" + sig.__str__() + ", " + eps.__str__() + ")...")
496         result = sdp.iterate()
497         end_time = time.time()
498         end_cpu = time.clock()
499         run_time = datetime.timedelta(seconds = int(end_time - start_time))
500         cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
501
502         point = Point(*([sig, eps] + key + [result, run_time, cpu_time]))
503         self.point_table.append(point)
504         self.save_point(point, self.name)
505
506     # Determines a full grid of Points.
507     # Appends the Points to point_table and the Grid to grid_table.
508     def determine_grid(self, key):
509         #if self.get_grid_index(key) != -1:
510         #start_time=time.time()
511         #start_cpu=time.clock()
512
513         grid = Grid(*(key + [[], [], 0, 0]))
514
515         self.determine_points(key)
516
517         # end_time=time.time()
518         # end_cpu=time.clock()
519         # run_time = datetime.timedelta(seconds = int(end_time - start_time))
520         # cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
521
522         points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax,
523             points.nmax] == key]
524         for point in points:
525             grid.run_time += point.run_time
526             grid.cpu_time += point.cpu_time
527             if point.allowed == True:
528                 grid.allowed_points.append((point.sig, point.eps))
529             else:
530                 grid.disallowed_points.append((point.sig, point.eps))
531
532         # grid.run_time = run_time
533         # grid.cpu_time = cpu_time
534         self.grid_table.append(grid)
535         # self.save_grid(grid, self.name)
536
537     # A method for composing a whole grid from a set of 'raw' points.
538     # Allows more flexibility - can choose sets of disparate points or use parallelization.
539     def make_grid(self, key):
540         grid = Grid(*(key + [[], [], 0, 0]))
541         points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax,
542             points.nmax] == key]
543         for point in points:

```

```

542     grid.run_time += point.run_time
543     grid.cpu_time += point.cpu_time
544     if point.allowed == True:
545         grid.allowed_points.append((point.sig, point.eps))
546     else:
547         grid.disallowed_points.append((point.sig, point.eps))
548
549 # A function used for the multi-correlator 3D Ising example.
550 # Note default is antisymmetrised convolved conformal blocks.
551     def convolved_table_list(self, tab1, tab2, tab3):
552         f_tab1a = bootstrap.ConvolvedBlockTable(tab1)
553         f_tab1s = bootstrap.ConvolvedBlockTable(tab1, symmetric = True)
554         f_tab2a = bootstrap.ConvolvedBlockTable(tab2)
555         f_tab2s = bootstrap.ConvolvedBlockTable(tab2, symmetric = True)
556         f_tab3 = bootstrap.ConvolvedBlockTable(tab3)
557         return [f_tab1a, f_tab1s, f_tab2a, f_tab2s, f_tab3]
558
559 # Returns the number of points that will be calculated for given sig,eps ranges and step sizes.
560     def points(self):
561         return ((self.sig_range[1] - self.sig_range[0])/self.sig_step) * ((self.eps_range[1] - self.
562             eps_range[0])/self.eps_step)
563
564 # Saves a Point object' data to file named in self.name
565     def save_point(self, point, name):
566         with open(name + ".py", 'a') as file:
567             file.write("kmax = " + str(point.kmax) + "\n")
568             file.write("lmax = " + str(point.lmax) + "\n")
569             file.write("mmax = " + str(point.mmax) + "\n")
570             file.write("nmax = " + str(point.nmax) + "\n")
571             file.write("sig = " + str(point.sig) + "\n")
572             file.write("eps = " + str(point.eps) + "\n")
573             file.write("allowed = " + str(point.allowed) + "\n")
574             file.write("run_time = " + str(point.run_time) + "\n")
575             file.write("cpu_time = " + str(point.cpu_time) + "\n")
576             file.write("self.point_table.append(Point(kmax, lmax, mmax, nmax, sig, eps, run_time,
577                 cpu_time))" + "\n")
578
579 # Recoveres a table of Point objects stored to a file.
580     def recover_points(self, file_name):
581         exec(open(file_name + ".py").read())

```