
new-ising-class.py

```
1  # We create a 'master' Ising class, with options to gap the spectrum or use mixed correlator  
   information.  
2  import subprocess  
3  import bootstrap  
4  import matplotlib.pyplot as plt  
5  import time, datetime  
6  import datetime  
7  import numpy as np  
8  from matplotlib.backends.backend_pdf import PdfPages  
9  
10 sig_defaults = np.arange(0.5,0.85,0.05).tolist()  
11 eps_defaults = np.arange(1.0,2.2,0.2).tolist()  
12  
13 class Point(object):  
14     def __init__(self, sig, eps, kmax, lmax, mmax, nmax, allowed, run_time, cpu_time, CB_time,  
        CB_cpu, xml_time, xml_cpu, sdp_time, sdp_cpu):  
15         self.sig = sig  
16         self.eps = eps  
17         self.kmax = kmax  
18         self.lmax = lmax  
19         self.mmax = mmax  
20         self.nmax = nmax  
21         self.allowed = allowed  
22         self.run_time = run_time  
23         self.cpu_time = cpu_time  
24         self.CB_time = CB_time  
25         self.CB_cpu = CB_cpu  
26         self.xml_time = xml_time  
27         self.xml_cpu = xml_cpu  
28         self.sdp_time = sdp_time  
29         self.sdp_cpu = sdp_cpu  
30  
31 # Saves a Point object' data to file named in self.name  
32     def save(self, name):  
33         with open(name + ".py", 'a') as file:  
34             file.write("sig = " + str(self.sig) + "\n")  
35             file.write("eps = " + str(self.eps) + "\n")  
36             file.write("kmax = " + str(self.kmax) + "\n")  
37             file.write("lmax = " + str(self.lmax) + "\n")  
38             file.write("mmax = " + str(self.mmax) + "\n")  
39             file.write("nmax = " + str(self.nmax) + "\n")  
40             file.write("allowed = " + str(self.allowed) + "\n")  
41             file.write("run_time = " + str(self.run_time) + "\n")  
42             file.write("cpu_time = " + str(self.cpu_time) + "\n")  
43             file.write("CB_time = " + str(self.CB_time) + "\n")  
44             file.write("CB_cpu = " + str(self.CB_cpu) + "\n")  
45             file.write("xml_time = " + str(self.xml_time) + "\n")  
46             file.write("xml_cpu = " + str(self.xml_cpu) + "\n")  
47             file.write("sdp_time = " + str(self.sdp_time) + "\n")  
48             file.write("sdp_cpu = " + str(self.sdp_cpu) + "\n")  
49             file.write("self.point_table.append(Point(sig, eps, kmax, lmax, mmax, nmax, allowed,  
                run_time, cpu_time, CB_time, CB_cpu, xml_time, xml_cpu, sdp_time, sdp_cpu))" + "\n")  
50  
51 class Grid(object):
```

```

52 def __init__(self, kmax, lmax, mmax, nmax, allowed_points, disallowed_points, run_time,
53             cpu_time):
54     self.kmax = kmax
55     self.lmax = lmax
56     self.mmax = mmax
57     self.nmax = nmax
58     self.allowed_points = allowed_points
59     self.disallowed_points = disallowed_points
60     self.run_time = run_time
61     self.cpu_time = cpu_time
62
63 def save(self, name):
64     with open(name + ".py", 'a') as file:
65         file.write("kmax = " + str(self.kmax) + "\n")
66         file.write("lmax = " + str(self.lmax) + "\n")
67         file.write("mmax = " + str(self.mmax) + "\n")
68         file.write("nmax = " + str(self.nmax) + "\n")
69         file.write("allowed_points = " + str(self.allowed_points) + "\n")
70         file.write("disallowed_points = " + str(self.disallowed_points) + "\n")
71         file.write("run_time = " + str(self.run_time) + "\n")
72         file.write("cpu_time = " + str(self.cpu_time) + "\n")
73         file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
74             disallowed_points, run_time, cpu_time))" + "\n")
75
76 class Ising(object):
77     def __init__(self, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults):
78         self.dim = dim
79         self.gap = gap
80         self.sig_values = sig_values
81         self.eps_values = eps_values
82         self.grid_table = []
83         self.grid_file = "grid_saves"
84         # self.name = name
85
86         # For a given set of conformal blocks, set by kmax and lmax, generate a grids for a specified
87         range of mmax and nmax.
88
89         # If we obtain a grid of entirely dissallowed points, fill in the rest of the grids for that
90         kmax and lmax.
91
92     def iterate_parameters(self, kmax_range, lmax_range, mmax_range, nmax_range):
93         keys = self.generate_keys(kmax_range, lmax_range, mmax_range, nmax_range)
94
95         while len(keys) > 0:
96             # Used keys will store the keys for which there is already a grid in table.
97             used_keys = []
98             #null_keys = []
99
100             for key in keys:
101                 if self.get_grid_index(key) != -1:
102                     used_keys.append(key)
103                     continue
104                 print("Trying kmax = " + str(key[0]) + ", lmax = " + str(key[1]) + ", mmax = " + str(key
105                     [2]) + ", nmax = " + str(key[3]))
106                 self.determine_grid(key)
107                 used_keys.append(key)
108
109             # If the grid has only disallowed points...
110             if self.grid_table[self.get_grid_index(key)].allowed_points == []:
111                 print ("In the if statement.")

```

```

105         k = key[0]
106         l = key[1]
107         m = key[2]
108         n = key[3]
109
110         null_keys = [key for key in keys if key not in used_keys and key[0] == k and key[1] ==
111                     l and key[2] >= m and key[3] >= n]
112
113         for key in null_keys:
114             if self.get_grid_index(key) != -1:
115                 used_keys.append(key)
116                 continue
117             #grid = Grid(*key, [], [])
118             grid = Grid(*(key + [[], [], 0, 0]))
119
120             for sig in self.sig_values:
121                 for eps in self.eps_values:
122                     grid.disallowed_points.append((sig, eps))
123
124             self.grid_table.append(grid)
125             grid.save(self.grid_file)
126             #self.save_grid(grid, self.name)
127
128         break
129
130     # We remove all keys from the list that we are done with.
131     keys = [key for key in keys if key not in null_keys and key not in used_keys]
132     null_keys = []
133
134     '''
135     # Saves the data as an executable file that will repopulate the table attribute.
136     # Note, we now do this as we go, instead of at the end, to avoid loss of mass data.
137     def save_to_file(self, name):
138         with open(name + ".py", 'w') as file:
139             file.write("self.table = []\n")
140             for grid in self.table:
141                 file.write("kmax = " + str(grid.kmax) + "\n")
142                 file.write("lmax = " + str(grid.lmax) + "\n")
143                 file.write("mmax = " + str(grid.mmax) + "\n")
144                 file.write("nmax = " + str(grid.nmax) + "\n")
145                 file.write("allowed_points = " + str(grid.allowed_points) + "\n")
146                 file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")
147                 file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
148                 disallowed_points))" + "\n")
149
150     '''
151
152     '''
153     def save_grid(self, grid, name):
154         with open(name + ".py", 'a') as file:
155             file.write("kmax = " + str(grid.kmax) + "\n")
156             file.write("lmax = " + str(grid.lmax) + "\n")
157             file.write("mmax = " + str(grid.mmax) + "\n")
158             file.write("nmax = " + str(grid.nmax) + "\n")
159             file.write("allowed_points = " + str(grid.allowed_points) + "\n")
160             file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")
161             file.write("run_time = " + str(grid.run_time) + "\n")
162             file.write("cpu_time = " + str(grid.cpu_time) + "\n")

```

```

161         file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
162             disallowed_points, run_time, cpu_time))" + "\n")
163     '''
164     # Recoveres a table stored to a file.
165     # Loads point_table's and grid_table's.
166     def load_table(self, file_name):
167         #exec(open(file_name + ".py").read())
168         with open(file_name + ".py") as infile:
169             for line in infile:
170                 exec(line)
171
172     # Searches table of grids for index matching the input key. Returns -1 if not found.
173     def get_grid_index(self, key):
174         for i in range(0, len(self.grid_table)):
175             if self.grid_table[i].kmax == key[0] and self.grid_table[i].lmax == key[1] and self.
176                 grid_table[i].mmax == key[2] and self.grid_table[i].nmax == key[3]:
177                 return i
178         return -1
179
180     # Plots a single grid, specified by a key. Note grid must be in grid_table.
181     def plot_grid(self, key):
182         grid = grid_table[self.get_grid_index(key)]
183         allowed_sig = [points[0] for points in grid.allowed_points]
184         allowed_eps = [points[1] for points in grid.allowed_points]
185         disallowed_sig = [points[0] for points in grid.disallowed_points]
186         disallowed_eps = [points[1] for points in grid.disallowed_points]
187
188         # Plot a grid.
189         plt.plot(allowed_sig, allowed_eps, 'r+')
190         plt.plot(disallowed_sig, disallowed_eps, 'b+')
191         plt.title('kmax : ' + grid.kmax.__str__() + " " +
192             'lmax : ' + grid.lmax.__str__() + " " +
193             'mmax : ' + grid.mmax.__str__() + " " +
194             'nmax : ' + grid.nmax.__str__())
195
196     # Plots and saves a series of grids to an output PDF file.
197     # Takes as input parameter values for which we want plotted grids, and the desired PDF file
198     # name.
199     def plot_grids(self, keys, file_name, plots_per_page, grid_size):
200         #tab = self.generate_table(keys)
201         #table = [grid for grid in tab if grid.run_time != 0]
202         table = self.grid_table
203         pdf_pages = PdfPages(file_name + ".pdf")
204
205         # Define the number of plots per page and the size of the grid board.
206         nb_plots = len(table)
207         # nb_plots_per_page = 6
208         nb_pages = int(np.ceil(nb_plots / float(plots_per_page)))
209         # grid_size=(3,2)
210
211         # This will define which row of the grid we are on.
212         row_index = 0
213
214         # We go through each 'grid' in 'grid_table', generating a plot for each.
215         for i in range(nb_plots):
216             # To begin, declare a new figure / page if we have exceeded limit of the last page.
217             if i % plots_per_page == 0:

```

```

216         fig = plt.figure(figsize=(8.27, 11.69), dpi=100)
217
218         # Now, add a plot for the current grid on the grid board.
219         plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
220         if i % grid_size[1] == 1:
221             row_index += 1
222
223         # Handle our data. Retrieve isolated points for plotting from our input grid_table of Grid
objects.
224         allowed_sig = [points[0] for points in table[i].allowed_points]
225         allowed_eps = [points[1] for points in table[i].allowed_points]
226         disallowed_sig = [points[0] for points in table[i].disallowed_points]
227         disallowed_eps = [points[1] for points in table[i].disallowed_points]
228
229         # Plot a grid.
230         # if table[i].run_time != 0 and table[i].cpu_time != 0:
231         plt.plot(allowed_sig, allowed_eps, 'r+')
232         plt.plot(disallowed_sig, disallowed_eps, 'b+')
233         plt.title('[' + table[i].kmax.__str__() + ", "
234                 + table[i].lmax.__str__() + ", "
235                 + table[i].mmax.__str__() + ", "
236                 + table[i].nmax.__str__() + ']'
237                 + " " + time.strftime('%H:%M:%S', table[i].run_time))
238         #else:
239         # plt.plot(allowed_sig, allowed_eps, 'r+')
240         # plt.plot(disallowed_sig, disallowed_eps, 'b+')
241         # plt.title('[' + table[i].kmax.__str__() + ", "
242         #         + table[i].lmax.__str__() + ", "
243         #         + table[i].mmax.__str__() + ", "
244         #         + table[i].nmax.__str__() + ']'
245         #         + " " + "AUTOFILLED")
246         #plt.title('kmax : ' + table[i].kmax.__str__() + " " +
247         # 'lmax : ' + table[i].lmax.__str__() + " " +
248         # 'mmax : ' + table[i].mmax.__str__() + " " +
249         # 'nmax : ' + table[i].nmax.__str__())
250
251         # If we have filled a page, or have reached the end of our plots, tight-pack and save the
page.
252         if (i + 1) % plots_per_page == 0 or (i + 1) == nb_plots:
253             plt.tight_layout()
254             pdf_pages.savefig(fig)
255             row_index = 0
256
257     pdf_pages.close()
258
259     # Returns a key or list of keys generated by the input parameter ranges.
260     def generate_keys(self, kmax_range, lmax_range, mmax_range, nmax_range):
261         if type(kmax_range) == int:
262             kmax_range = [kmax_range]
263         if type(lmax_range) == int:
264             lmax_range = [lmax_range]
265         if type(mmax_range) == int:
266             mmax_range = [mmax_range]
267         if type(nmax_range) == int:
268             nmax_range = [nmax_range]
269         keys = []
270         for kmax in kmax_range:
271             for lmax in lmax_range:

```

```

272         for mmax in mmax_range:
273             for nmax in nmax_range:
274                 key = [kmax, lmax, mmax, nmax]
275                 keys.append(key)
276     return keys
277
278     # Generates a subtable table of desired, already determined grids from main table.
279     # Gives a warning message if a grid isn't found.
280     def generate_table(self, keys):
281         # table to store the resulting grids.
282         table = []
283         for key in keys:
284             if self.get_grid_index(key) == -1:
285                 print("Grid at kmax = " + str(key[0]) + ", " +
286                       "lmax = " + str(key[1]) + ", " +
287                       "mmax = " + str(key[2]) + ", " +
288                       "nmax = " + str(key[3]) + ", " + "does not exist.")
289             else:
290                 table.append(self.grid_table[self.get_grid_index(key)])
291
292     return table
293
294     # Takes two keys and returns a dictionary with the direction of every point.
295     def changes(self, key1, key2):
296         changes = {}
297         allowed_one = self.grid_table[self.get_grid_index(key1)].allowed_points
298         allowed_two = self.grid_table[self.get_grid_index(key2)].allowed_points
299
300         for sig in self.sig_values:
301             for eps in self.eps_values:
302                 if (sig, eps) in allowed_one and (sig, eps) in allowed_two:
303                     changes[(sig, eps)] = 0
304                 if (sig, eps) not in allowed_one and (sig, eps) not in allowed_two:
305                     changes[(sig, eps)] = 0
306                 if (sig, eps) in allowed_one and (sig, eps) not in allowed_two:
307                     changes[(sig, eps)] = -1
308                 if (sig, eps) not in allowed_one and (sig, eps) in allowed_two:
309                     changes[(sig, eps)] = 1
310         return changes
311
312     # grid_size is a tuple of (rows, columns).
313     def plot_changes(self, keys, file_name, plots_per_page, grid_size):
314         pdf_pages = PdfPages(file_name + ".pdf")
315
316         # Define the number of plots per page and the size of the grid board.
317         # We have one less plots than grids.
318         nb_plots = len(keys)
319         # nb_plots_per_page = 6
320         nb_pages = int(np.ceil(nb_plots / float(plots_per_page)))
321         # grid_size=(3,2)
322
323         # This will define which row of the grid we are on.
324         row_index = 0
325
326         # We go through each 'grid' in 'grid_table', generating a plot for each.
327         for i in range(nb_plots):
328             # To begin, declare a new figure / page if we have exceeded limit of the last page.
329             # 8.27 x 11.69 dimensions of A4 page in inches. DPI - dots per inch (resolution.)

```

```

330 if i % plots_per_page == 0:
331     fig = plt.figure(figsize=(8.27, 11.69), dpi=100)
332
333     # Now, add a plot for the current grid on the grid board.
334     plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
335     if i % grid_size[1] == 1:
336         row_index += 1
337
338     # We want the first grid to compare all changes to.
339     if i == 0:
340         grid = self.grid_table[self.get_grid_index(keys[i])]
341         allowed_sig = [points[0] for points in grid.allowed_points]
342         allowed_eps = [points[1] for points in grid.allowed_points]
343         disallowed_sig = [points[0] for points in grid.disallowed_points]
344         disallowed_eps = [points[1] for points in grid.disallowed_points]
345
346         # Plot the grid.
347         plt.plot(allowed_sig, allowed_eps, 'r+')
348         plt.plot(disallowed_sig, disallowed_eps, 'b+')
349         #plt.title('kmax : ' + grid.kmax.__str__() + " " +
350         # 'lmax : ' + grid.lmax.__str__() + " " +
351         # 'mmax : ' + grid.mmax.__str__() + " " +
352         # 'nmax : ' + grid.nmax.__str__())
353         if table[i].run_time != 0 and table[i].cpu_time != 0:
354             plt.plot(allowed_sig, allowed_eps, 'r+')
355             plt.plot(disallowed_sig, disallowed_eps, 'b+')
356             plt.title('[' + table[i].kmax.__str__() + ", "
357                     + table[i].lmax.__str__() + ", "
358                     + table[i].mmax.__str__() + ", "
359                     + table[i].nmax.__str__() + ']'
360                     + " " + time.strftime('%H:%M:%S', table[i].run_time))
361         #else:
362         # plt.plot(allowed_sig, allowed_eps, 'r+')
363         # plt.plot(disallowed_sig, disallowed_eps, 'b+')
364         # plt.title('[' + table[i].kmax.__str__() + ", "
365         # + table[i].lmax.__str__() + ", "
366         # + table[i].mmax.__str__() + ", "
367         # + table[i].nmax.__str__() + ']'
368         # + " " + "AUTOFILLED")
369
370         y_range = plt.ylim()
371         x_range = plt.xlim()
372
373     else:
374         changes = self.changes(keys[i-1], keys[i])
375         unchanged_points = []
376         to_allowed_points = []
377         to_disallowed_points = []
378         for point in changes:
379             if changes[point] == 0:
380                 unchanged_points.append(point)
381             if changes[point] == 1:
382                 to_allowed_points.append(point)
383             if changes[point] == -1:
384                 to_disallowed_points.append(point)
385
386         unchanged_sig = [points[0] for points in unchanged_points]
387         unchanged_eps = [points[1] for points in unchanged_points]

```

```

388     to_disallowed_sig = [points[0] for points in to_disallowed_points]
389     to_disallowed_eps = [points[1] for points in to_disallowed_points]
390     to_allowed_sig = [points[0] for points in to_allowed_points]
391     to_allowed_eps = [points[1] for points in to_allowed_points]
392
393     # Plot a grid.
394     plt.plot(to_allowed_sig, to_allowed_eps, 'r+')
395     plt.plot(to_disallowed_sig, to_disallowed_eps, 'b+')
396     plt.xlim(x_range)
397     plt.ylim(y_range)
398     plt.title('kmax : ' + self.grid_table[self.get_grid_index(keys[i])].kmax.__str__() + " "
399             +
400             'lmax : ' + self.grid_table[self.get_grid_index(keys[i])].lmax.__str__() + " " +
401             'mmax : ' + self.grid_table[self.get_grid_index(keys[i])].mmax.__str__() + " " +
402             'nmax : ' + self.grid_table[self.get_grid_index(keys[i])].nmax.__str__())
403
404     # If we have filled a page, or have reached the end of our plots, tight-pack and save the
405     page.
406     if (i + 1) % plots_per_page == 0 or (i + 1) == nb_plots:
407         plt.tight_layout()
408         pdf_pages.savefig(fig)
409         row_index = 0
410
411     pdf_pages.close()
412
413 class SingleCorrelator(Ising):
414     bootstrap.cutoff=1e-10
415     def __init__(self, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults):
416         self.dim = dim
417         self.gap = gap
418         self.sig_values = sig_values
419         self.eps_values = eps_values
420         self.grid_table = []
421         self.grid_file = "grid_saves"
422
423     # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
424     def determine_grid(self, key):
425         #if self.get_grid_index(key) != -1:
426         start_time=time.time()
427         start_cpu=time.clock()
428         tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
429         tab2 = bootstrap.ConvolvedBlockTable(tab1)
430
431         # Instantiate a Grid object with appropriate input values.
432         # grid=Grid(*key, [], [])
433         grid = Grid(*(key + [[], [], 0, 0]))
434
435     for sig in self.sig_values:
436         for eps in self.eps_values:
437             sdp = bootstrap.SDP(sig, tab2)
438             # SDPB will naturally try to parallelize across 4 cores / slots.
439             # To prevent this, we set its 'maxThreads' option to 1.
440             # See 'common.py' for the list of SDPB option strings, as well as their default values.
441             sdp.set_option("maxThreads", 1)
442             sdp.set_bound(0, float(self.gap))
443             sdp.add_point(0, eps)
444             result = sdp.iterate()
445             if result:

```



```

444         grid.allowed_points.append((sig, eps))
445     else:
446         grid.disallowed_points.append((sig, eps))
447
448     # Now append this grid object to the IsingGap grid_table.
449     # Note we will need to implement a look up table to retrieve desired data.
450     end_time=time.time()
451     end_cpu=time.clock()
452     run_time=end_time-start_time
453     cpu_time=end_cpu-start_cpu
454     run_time = datetime.timedelta(seconds = int(end_time - start_time))
455     cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
456
457     grid.run_time = run_time
458     grid.cpu_time = cpu_time
459     self.grid_table.append(grid)
460     grid.save(self.grid_file)
461     #self.save_grid(grid, self.name)
462
463     # For mixed correlator, we pass pairs of external scaling dimensions to the SDP.
464     # We copy the content of the triples entering the SDP from the tutorial, same case.
465     # We want to scan over all possible [sig, eps], assuming only one relevant Z2-even and Z2-odd
operator.
466     # Use a prototype to use the same basis for all SDPs, so we don't need to recalculate bases.
467     # Dump the ConformalBlockTable objects once we have used them to save memory.
468     # Set dualThresholdError to 1e-15.
469     # Use 16 cores for all SDP runs - set maxThreads = 16, speed up the SDP.
470     class MixedCorrelator(Ising):
471         bootstrap.cutoff=0
472         def __init__(self, dim = 3):
473             self.dim = dim
474             self.point_table = []
475             self.grid_table = []
476             self.grid_file = "grid_saves"
477             self.point_file = "point_saves"
478
479     # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
480     def determine_points(self, key, row):
481     # Will be called with a given row_lists[i]
482     # row = row_lists[row_index]
483     reference_sdp = None
484     blocks_initiated = False
485     for i in range(len(row[0])):
486         sig = row[0][i]
487         eps = row[1][i]
488
489     global start_time
490     start_time = time.time()
491     global start_cpu
492     start_cpu = time.clock()
493     # Generate three conformal block tables, two of which depend on the dimension differences.
494     # They need only be calculated once for any given diagonal. They remain constant along this
line.
495     # Uses the function above to return the 5 ConvolvedConformalBlocks we need.
496     # The ConvolvedConformalBlock objects inherits the dimension differences from
ConformalBlockTable.
497     # We set odd_spins = True for odd those ConvolvedConformalBlocks appearing in odd-sector-
odd-spins.

```

```

498 # We set symmetric = True where required.
499 if blocks_initiated == False:
500     g_tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
501     g_tab2 = bootstrap.ConformalBlockTable(self.dim, *(key + [eps-sig, sig-eps, "odd_spins =
        True"]))
502     g_tab3 = bootstrap.ConformalBlockTable(self.dim, *(key + [sig-eps, sig-eps, "odd_spins =
        True"]))
503     tab_list = self.convolved_table_list(g_tab1, g_tab2, g_tab3)
504     for tab in [g_tab1, g_tab2, g_tab3]:
505         tab.dump("tab_" + str(tab.delta_12) + "_" + str(tab.delta_34))
506         del tab
507     blocks_initiated = True
508 global now
509 global now_clock
510 global CB_time
511 global CB_cpu
512 now = time.time()
513 now_clock = time.clock()
514 CB_time = datetime.timedelta(seconds = int(now - start_time))
515 CB_cpu = datetime.timedelta(seconds = int(now_clock - start_cpu))
516 print("The calculation of the required conformal blocks has successfully completed.")
517 print("Time taken: " + str(CB_time))
518 print("CPU_time: " + str(CB_cpu))
519 # N.B vec3 & vec2 are 'raw' quads, which will be converted to 1x1 matrices automatically.
520 # 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
521 vec3 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [-1, 2, 0, 0], [1, 3, 0, 0]]
522 # 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
523 vec2 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [1, 2, 0, 0], [-1, 3, 0, 0]]
524 # The first vector has five components as well but they are matrices of quads, not just the
        quads themselves.
525 m1 = [[[1, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
526 m2 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [1, 0, 1, 1]]]
527 m3 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
528 m4 = [[[0, 0, 0, 0], [0.5, 0, 0, 1]], [[0.5, 0, 0, 1], [0, 0, 0, 0]]]
529 m5 = [[[0, 1, 0, 0], [0.5, 1, 0, 1]], [[0.5, 1, 0, 1], [0, 1, 0, 0]]]
530 vec1 = [m1, m2, m3, m4, m5]
531
532 # The first rep must be the singlet even channel, where the unit operator resides.
533 # After this, the order doesn't matter.
534 # Spins for these again go even, even, odd.
535 # The Z2 even sector has only even spins, Z2 odd sector runs over even and odd spins.
536 info = [[vec1, 0, "z2-even-l-even"], [vec2, 0, "z2-odd-l-even"], [vec3, 1, "z2-odd-l-odd"]]
537
538 # We instantiate the SDP object, inputting our vectorial sum info.
539 # dim_list, convolved_block_table_list, vector_types (how they combine to compose sum rule)
        .
540 # We use the first calculated SDP object as a prototype for all the rest.
541 # This is because some bounds remain unchanged, no need to recalculate basis.
542 # Basis is independent of external scaling dimensions, cares only of the bounds on
        particular operators.
543 # sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
544 if reference_sdp == None:
545     sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
546     reference_sdp = sdp
547 else:
548     sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info, prototype = reference_sdp)

```

```

549
550 # We assume the continuum in both Z2 odd / even sectors begins at the dimension=3.
551 sdp.set_bound([0, "z2-even-l-even"], self.dim)
552 sdp.set_bound([0, "z2-odd-l-even"], self.dim)
553
554 # Except for the two lowest dimension scalar operators in each sector.
555 sdp.add_point([0, "z2-even-l-even"], eps)
556 sdp.add_point([0, "z2-odd-l-even"], sig)
557
558 # We expect these calculations to be computationally intensive.
559 # We set maxThreads=16 to parallelise SDPB for all runs.
560 # See 'common.py' for the list of SDPB option strings, as well as their default values.
561 sdp.set_option("maxThreads", 16)
562 sdp.set_option("dualErrorThreshold", 1e-15)
563 sdp.set_option("maxIterations", 1000)
564
565 # Run the SDP to determine if the current operator spectrum is permissible.
566 print("Testing point " + "(" + sig.__str__() + ", " + eps.__str__() + "...")
567 result = sdp.iterate()
568 end_time = time.time()
569 end_cpu = time.clock()
570 global sdp_time
571 global sdp_cpu
572 sdp_time = datetime.timedelta(seconds = int(end_time - bootstrap.now2))
573 sdp_cpu = datetime.timedelta(seconds = int(end_cpu - bootstrap.now2_clock))
574 run_time = datetime.timedelta(seconds = int(end_time - start_time))
575 cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
576
577 print("The SDP has finished running.")
578 print("Time taken: " + str(sdp_time))
579 print("CPU_time: " + str(sdp_cpu))
580 print("See point file for more information. Check the times are consistent")
581
582 point = Point(*([sig, eps] + key + [result, run_time, cpu_time, CB_time, CB_cpu, bootstrap.
583               xml_time, bootstrap.xml_cpu, sdp_time, sdp_cpu]))
584 self.point_table.append(point)
585 point.save(self.point_file)
586 #self.save_point(point, self.name)
587
588 # Determines a full grid of Points.
589 # Appends the Points to point_table and the Grid to grid_table.
590 def determine_grid(self, key):
591     #if self.get_grid_index(key) != -1:
592     #start_time=time.time()
593     #start_cpu=time.clock()
594
595     grid = Grid(*(key + [[], [], 0, 0]))
596
597     self.determine_points(key)
598
599     # end_time=time.time()
600     # end_cpu=time.clock()
601     # run_time = datetime.timedelta(seconds = int(end_time - start_time))
602     # cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
603
604     points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax,
605               points.nmax] == key]
606     for point in points:

```

```

605     grid.run_time += point.run_time
606     grid.cpu_time += point.cpu_time
607     if point.allowed == True:
608         grid.allowed_points.append((point.sig, point.eps))
609     else:
610         grid.disallowed_points.append((point.sig, point.eps))
611
612     # grid.run_time = run_time
613     # grid.cpu_time = cpu_time
614     self.grid_table.append(grid)
615     # self.save_grid(grid, self.name)
616
617     # A method for composing a whole grid from a set of 'raw' points.
618     # Allows more flexibility - can choose sets of disparate points or use parallelization.
619     def make_grid(self, key):
620         grid = Grid(*(key + [[], [], 0, 0]))
621         points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax,
622             points.nmax] == key]
623         for point in points:
624             grid.run_time += point.run_time
625             grid.cpu_time += point.cpu_time
626             if point.allowed == True:
627                 grid.allowed_points.append((point.sig, point.eps))
628             else:
629                 grid.disallowed_points.append((point.sig, point.eps))
630
631     # A function used for the multi-correlator 3D Ising example.
632     # Note default is antisymmetrised convolved conformal blocks.
633     def convolved_table_list(self, tab1, tab2, tab3):
634         f_tab1a = bootstrap.ConvolvedBlockTable(tab1)
635         f_tab1s = bootstrap.ConvolvedBlockTable(tab1, symmetric = True)
636         f_tab2a = bootstrap.ConvolvedBlockTable(tab2)
637         f_tab2s = bootstrap.ConvolvedBlockTable(tab2, symmetric = True)
638         f_tab3 = bootstrap.ConvolvedBlockTable(tab3)
639         return [f_tab1a, f_tab1s, f_tab2a, f_tab2s, f_tab3]
640
641     # Returns the number of points that will be calculated for given sig,eps ranges and step sizes.
642     def points(self):
643         return ((self.sig_range[1] - self.sig_range[0])/self.sig_step) * ((self.eps_range[1] - self.
644             eps_range[0])/self.eps_step)
645     '''
646
647     # Saves a Point object' data to file named in self.name
648     def save_point(self, point, name):
649         with open(name + ".py", 'a') as file:
650             file.write("kmax = " + str(point.kmax) + "\n")
651             file.write("lmax = " + str(point.lmax) + "\n")
652             file.write("mmax = " + str(point.mmax) + "\n")
653             file.write("nmax = " + str(point.nmax) + "\n")
654             file.write("sig = " + str(point.sig) + "\n")
655             file.write("eps = " + str(point.eps) + "\n")
656             file.write("allowed = " + str(point.allowed) + "\n")
657             file.write("run_time = " + str(point.run_time) + "\n")
658             file.write("cpu_time = " + str(point.cpu_time) + "\n")
659             file.write("self.point_table.append(Point(kmax, lmax, mmax, nmax, sig, eps, run_time,
660                 cpu_time))" + "\n")
661     '''
662
663     # Recoveres a table of Point objects stored to a file.
664     def recover_points(self, file_name):

```

```
660 | with open(file_name + ".py") as infile:
661 |     for line in infile:
662 |         exec(line)
663 | #exec(open(file_name + ".py").read())
```