
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 Grid(object):  
13     def __init__(self, kmax, lmax, mmax, nmax, allowed_points, disallowed_points, run_time,  
   cpu_time):  
14         self.kmax = kmax  
15         self.lmax = lmax  
16         self.mmax = mmax  
17         self.nmax = nmax  
18         self.allowed_points = allowed_points  
19         self.disallowed_points = disallowed_points  
20         self.run_time = run_time  
21         self.cpu_time = cpu_time  
22  
23 class Ising(object):  
24     def __init__(self, dim = 3, gap = 3, name, sig_values = sig_defaults, eps_values = eps_defaults  
   ):  
25         self.dim = dim  
26         self.gap = gap  
27         self.sig_values = sig_values  
28         self.eps_values = eps_values  
29         self.table = []  
30         self.name = name  
31  
32     # For a given set of conformal blocks, set by kmax and lmax, generate a grids for a specified  
   range of mmax and nmax.  
33     # If we obtain a grid of entirely disallowed points, fill in the rest of the grids for that  
   kmax and lmax.  
34     def iterate_parameters(self, kmax_range, lmax_range, mmax_range, nmax_range):  
35         keys = self.generate_keys(kmax_range, lmax_range, mmax_range, nmax_range)  
36  
37         while len(keys) > 0:  
38             # Used keys will store the keys for which there is already a grid in table.  
39             used_keys = []  
40             #null_keys = []  
41  
42             for key in keys:  
43                 if self.get_grid_index(key) != -1:  
44                     used_keys.append(key)  
45                     continue  
46                 print("Trying kmax = " + str(key[0]) + ", lmax = " + str(key[1]) + ", mmax = " + str(key  
   [2]) + ", nmax = " + str(key[3]))  
47                 self.determine_grid(key)  
48                 used_keys.append(key)
```

```

49
50 # If the grid has only disallowed points...
51 if self.table[self.get_grid_index(key)].allowed_points == []:
52     print ("In the if statement.")
53     k = key[0]
54     l = key[1]
55     m = key[2]
56     n = key[3]
57
58     null_keys = [key for key in keys if key not in used_keys and key[0] == k and key[1] ==
        l and key[2] >= m and key[3] >= n]
59
60     for key in null_keys:
61         if self.get_grid_index(key) != -1:
62             used_keys.append(key)
63             continue
64         #grid = Grid(*key, [], [])
65         grid = Grid(*(key + [[], [], 0, 0]))
66
67         for sig in self.sig_values:
68             for eps in self.eps_values:
69                 grid.disallowed_points.append((sig, eps))
70
71         self.table.append(grid)
72         self.save_grid(grid, self.name)
73
74     break
75
76 # We remove all keys from the list that we are done with.
77 keys = [key for key in keys if key not in null_keys and key not in used_keys]
78 null_keys = []
79
80
81 '''
82 # Saves the data as an executable file that will repopulate the table attribute.
83 # Note, we now do this as we go, instead of at the end, to avoid loss of mass data.
84 def save_to_file(self, name):
85     with open(name + ".py", 'w') as file:
86         file.write("self.table = []\n")
87         for grid in self.table:
88             file.write("kmax = " + str(grid.kmax) + "\n")
89             file.write("lmax = " + str(grid.lmax) + "\n")
90             file.write("mmax = " + str(grid.mmax) + "\n")
91             file.write("nmax = " + str(grid.nmax) + "\n")
92             file.write("allowed_points = " + str(grid.allowed_points) + "\n")
93             file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")
94             file.write("self.table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
                disallowed_points))" + "\n")
95
96 '''
97 def save_grid(self, grid, name):
98     with open(name + ".py", 'a') as file:
99         file.write("kmax = " + str(grid.kmax) + "\n")
100         file.write("lmax = " + str(grid.lmax) + "\n")
101         file.write("mmax = " + str(grid.mmax) + "\n")
102         file.write("nmax = " + str(grid.nmax) + "\n")
103         file.write("allowed_points = " + str(grid.allowed_points) + "\n")
104         file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")

```

```

105     file.write("run_time = " + str(grid.run_time) + "\n")
106     file.write("cpu_time = " + str(grid.cpu_time) + "\n")
107     file.write("self.table.append(Grid(kmax, lmax, mmax, nmax, allowed_points,
        disallowed_points, run_time, cpu_time))" + "\n")
108
109     # Recovers a table stored to a file.
110     def recover_table(self, file_name):
111         exec(open(file_name + ".py").read())
112
113
114     # Searches table of grids for index matching the input key. Returns -1 if not found.
115     def get_grid_index(self, key):
116         for i in range(0, len(self.table)):
117             if self.table[i].kmax == key[0] and self.table[i].lmax == key[1] and self.table[i].mmax ==
                key[2] and self.table[i].nmax == key[3]:
118                 return i
119         return -1
120
121     # Plots and saves a series of grids to an output PDF file.
122     # Takes as input parameter values for which we want plotted grids, and the desired PDF file
        name.
123     def plot_grids(self, keys, file_name):
124         table = self.generate_table(keys)
125         pdf_pages = PdfPages(file_name + ".pdf")
126
127         # Define the number of plots per page and the size of the grid board.
128         nb_plots = len(table)
129         nb_plots_per_page = 6
130         nb_pages = int(np.ceil(nb_plots / float(nb_plots_per_page)))
131         grid_size=(3,2)
132
133         # This will define which row of the grid we are on.
134         row_index = 0
135
136         # We go through each 'grid' in 'table', generating a plot for each.
137         for i in range(nb_plots):
138             # To begin, declare a new figure / page if we have exceeded limit of the last page.
139             if i % nb_plots_per_page == 0:
140                 fig = plt.figure(figsize=(8.27, 11.69), dpi=100)
141
142             # Now, add a plot for the current grid on the grid board.
143             plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
144             if i % grid_size[1] == 1:
145                 row_index += 1
146
147             # Handle our data. Retrieve isolated points for plotting from out input table of Grid
                objects.
148             allowed_sig = [points[0] for points in table[i].allowed_points]
149             allowed_eps = [points[1] for points in table[i].allowed_points]
150             disallowed_sig = [points[0] for points in table[i].disallowed_points]
151             disallowed_eps = [points[1] for points in table[i].disallowed_points]
152
153             # Plot a grid.
154             plt.plot(allowed_sig, allowed_eps, 'r+')
155             plt.plot(disallowed_sig, disallowed_eps, 'b+')
156             plt.title('kmax : ' + table[i].kmax.__str__() + " " +
                'lmax : ' + table[i].lmax.__str__() + " " +
                'mmax : ' + table[i].mmax.__str__() + " " +

```

```

159         'nmax : ' + table[i].nmax.__str__())
160
161     # If we have filled a page, or have reached the end of our plots, tight-pack and save the
162     # page.
163     if (i + 1) % nb_plots_per_page == 0 or (i + 1) == nb_plots:
164         plt.tight_layout()
165         pdf_pages.savefig(fig)
166         row_index = 0
167
168 pdf_pages.close()
169
170 # Returns a key or list of keys generated by the input parameter ranges.
171 def generate_keys(self, kmax_range, lmax_range, mmax_range, nmax_range):
172     if type(kmax_range) == int:
173         kmax_range = [kmax_range]
174     if type(lmax_range) == int:
175         lmax_range = [lmax_range]
176     if type(mmax_range) == int:
177         mmax_range = [mmax_range]
178     if type(nmax_range) == int:
179         nmax_range = [nmax_range]
180     keys = []
181     for kmax in kmax_range:
182         for lmax in lmax_range:
183             for mmax in mmax_range:
184                 for nmax in nmax_range:
185                     key = [kmax, lmax, mmax, nmax]
186                     keys.append(key)
187
188     return keys
189
190 # Generates a subtable table of desired, already determined grids from main table.
191 # Gives a warning message if a grid isn't found.
192 def generate_table(self, keys):
193     # table to store the resulting grids.
194     table = []
195     for key in keys:
196         if self.get_grid_index(key) == -1:
197             print("Grid at kmax = " + str(key[0]) + ", " +
198                   "lmax = " + str(key[1]) + ", " +
199                   "mmax = " + str(key[2]) + ", " +
200                   "nmax = " + str(key[3]) + ", " + "does not exist.")
201         else:
202             table.append(self.table[self.get_grid_index(key)])
203
204     return table
205
206 # Takes two keys and returns a dictionary with the direction of every point.
207 def changes(self, key1, key2):
208     changes = {}
209     allowed_one = self.table[self.get_grid_index(key1)].allowed_points
210     allowed_two = self.table[self.get_grid_index(key2)].allowed_points
211
212     for sig in self.sig_values:
213         for eps in self.eps_values:
214             if (sig, eps) in allowed_one and (sig, eps) in allowed_two:
215                 changes[(sig, eps)] = 0
216             if (sig, eps) not in allowed_one and (sig, eps) not in allowed_two:
217                 changes[(sig, eps)] = 0

```

```

216         if (sig, eps) in allowed_one and (sig, eps) not in allowed_two:
217             changes[(sig, eps)] = -1
218         if (sig, eps) not in allowed_one and (sig, eps) in allowed_two:
219             changes[(sig, eps)] = 1
220     return changes
221
222 def plot_changes(self, keys, file_name):
223     pdf_pages = PdfPages(file_name + ".pdf")
224
225     # Define the number of plots per page and the size of the grid board.
226     # We have one less plots than grids.
227     nb_plots = len(keys)
228     nb_plots_per_page = 6
229     nb_pages = int(np.ceil(nb_plots / float(nb_plots_per_page)))
230     grid_size=(3,2)
231
232     # This will define which row of the grid we are on.
233     row_index = 0
234
235     # We go through each 'grid' in 'table', generating a plot for each.
236     for i in range(nb_plots):
237         # To begin, declare a new figure / page if we have exceeded limit of the last page.
238         # 8.27 x 11.69 dimensions of A4 page in inches. DPI - dots per inch (resolution.)
239         if i % nb_plots_per_page == 0:
240             fig = plt.figure(figsize=(8.27, 11.69), dpi=100)
241
242         # Now, add a plot for the current grid on the grid board.
243         plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
244         if i % grid_size[1] == 1:
245             row_index += 1
246
247         # We want the first grid to compare all changes to.
248         if i == 0:
249             grid = self.table[self.get_grid_index(keys[i])]
250             allowed_sig = [points[0] for points in grid.allowed_points]
251             allowed_eps = [points[1] for points in grid.allowed_points]
252             disallowed_sig = [points[0] for points in grid.disallowed_points]
253             disallowed_eps = [points[1] for points in grid.disallowed_points]
254
255             # Plot the grid.
256             plt.plot(allowed_sig, allowed_eps, 'r+')
257             plt.plot(disallowed_sig, disallowed_eps, 'b+')
258             plt.title('kmax : ' + grid.kmax.__str__() + " " +
259                     'lmax : ' + grid.lmax.__str__() + " " +
260                     'mmax : ' + grid.mmax.__str__() + " " +
261                     'nmax : ' + grid.nmax.__str__())
262
263             y_range = plt.ylim()
264             x_range = plt.xlim()
265
266         else:
267             changes = self.changes(keys[i-1], keys[i])
268             unchanged_points = []
269             to_allowed_points = []
270             to_disallowed_points = []
271             for point in changes:
272                 if changes[point] == 0:
273                     unchanged_points.append(point)

```

```

274         if changes[point] == 1:
275             to_allowed_points.append(point)
276         if changes[point] == -1:
277             to_disallowed_points.append(point)
278
279     unchanged_sig = [points[0] for points in unchanged_points]
280     unchanged_eps = [points[1] for points in unchanged_points]
281     to_disallowed_sig = [points[0] for points in to_disallowed_points]
282     to_disallowed_eps = [points[1] for points in to_disallowed_points]
283     to_allowed_sig = [points[0] for points in to_allowed_points]
284     to_allowed_eps = [points[1] for points in to_allowed_points]
285
286     # Plot a grid.
287     plt.plot(to_allowed_sig, to_allowed_eps, 'r+')
288     plt.plot(to_disallowed_sig, to_disallowed_eps, 'b+')
289     plt.xlim(x_range)
290     plt.ylim(y_range)
291     plt.title('kmax : ' + self.table[self.get_grid_index(keys[i])].kmax.__str__() + " " +
292             'lmax : ' + self.table[self.get_grid_index(keys[i])].lmax.__str__() + " " +
293             'mmax : ' + self.table[self.get_grid_index(keys[i])].mmax.__str__() + " " +
294             'nmax : ' + self.table[self.get_grid_index(keys[i])].nmax.__str__())
295
296     # If we have filled a page, or have reached the end of our plots, tight-pack and save the
297     # page.
298     if (i + 1) % nb_plots_per_page == 0 or (i + 1) == nb_plots:
299         plt.tight_layout()
300         pdf_pages.savefig(fig)
301         row_index = 0
302
303     pdf_pages.close()
304
305 class SingleCorrelator(Ising):
306     bootstrap.cutoff=1e-10
307     def __init__(self, dim = 3, gap = 3, name, sig_values = sig_defaults, eps_values = eps_defaults
308         ):
309         self.dim = dim
310         self.gap = gap
311         self.sig_values = sig_values
312         self.eps_values = eps_values
313         self.table = []
314         self.name = name
315
316     # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
317     def determine_grid(self, key):
318         #if self.get_grid_index(key) != -1:
319         start_time=time.time()
320         start_cpu=time.clock()
321         tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
322         tab2 = bootstrap.ConvolvedBlockTable(tab1)
323
324         # Instantiate a Grid object with appropriate input values.
325         # grid=Grid(*key, [], [])
326         grid = Grid(*(key + [[], [], 0, 0]))
327
328     for sig in self.sig_values:
329         for eps in self.eps_values:
330             sdp = bootstrap.SDP(sig, tab2)
331             # SDPB will naturally try to parallelize across 4 cores / slots.

```

```

330     # To prevent this, we set its 'maxThreads' option to 1.
331     # See 'common.py' for the list of SDPB option strings, as well as their default values.
332     sdp.set_option("maxThreads", 1)
333     sdp.set_bound(0, float(self.gap))
334     sdp.add_point(0, eps)
335     result = sdp.iterate()
336     if result:
337         grid.allowed_points.append((sig, eps))
338     else:
339         grid.disallowed_points.append((sig, eps))
340
341     # Now append this grid object to the IsingGap table.
342     # Note we will need to implement a look up table to retrieve desired data.
343     end_time=time.time()
344     end_cpu=time.clock()
345     run_time=end_time-start_time
346     cpu_time=end_cpu-start_cpu
347     run_time = datetime.timedelta(seconds = int(end_time - start_time))
348     cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))
349
350     grid.run_time = run_time
351     grid.cpu_time = cpu_time
352     self.table.append(grid)
353     self.save_grid(grid, self.name)
354
355     # For mixed correlator, we pass pairs of external scaling dimensions to the SDP.
356     # We copy the content of the triples entering the SDP from the tutorial, same case.
357     # We want to scan over all possible [sig, eps], assuming only one relevant Z2-even and Z2-odd
358     # operator.
359     # Use a prototype to use the same basis for all SDPs, so we don't need to recalculate bases.
360     # Dump the ConformalBlockTable objects once we have used them to save memory.
361     # Set dualThresholdError to 1e-15.
362     # Use 16 cores for all SDP runs - set maxThreads = 16, speed up the SDP.
363     class MixedCorrelator(Ising):
364         bootstrap.cutoff=0
365         def __init__(self, dim = 3, gap = 3, name, sig_values = sig_defaults, eps_values = eps_defaults
366             ):
367             self.dim = dim
368             self.gap = gap
369             self.sig_values = sig_values
370             self.eps_values = eps_values
371             self.table = []
372             self.name = name
373
374     # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
375     def determine_grid(self, key):
376         #if self.get_grid_index(key) != -1:
377         reference_sdp = None
378         start_time=time.time()
379         start_cpu=time.clock()
380
381         # Instantiate a Grid object with appropriate input values.
382         # grid=Grid(*key, [], [])
383         grid = Grid(*(key + [[], [], 0, 0]))
384
385         for sig in self.sig_values:
386             for eps in self.eps_values:
387                 # Generates three tables, two of which depend on the dimension differences.

```

```

386 g_tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
387 g_tab2 = bootstrap.ConformalBlockTable(self.dim, *key, eps-sig, sig-eps, odd_spins =
      True)
388 g_tab3 = bootstrap.ConformalBlockTable(self.dim, *key, sig-eps, sig-eps, odd_spins =
      True)
389 # Uses the function above to return the 5 ConvolvedConformalBlocks we need.
390 # The ConvolvedConformalBlock objects inherits the dimension differences from
      ConformalBlockTable.
391 # We set odd_spins = True for odd those ConvolvedConformalBlocks appearing in odd-
      sector-odd-spins.
392 # We set symmetric = True where required.
393 tab_list = convolved_table_list(g_tab1, g_tab2, g_tab3)
394 # Here, we will save and delete conformal blocks. Think about their recycling.
395 # Otherwise, massive redundancy, same blocks will be used many times.
396
397 # N.B vec3 & vec2 are 'raw' quads, which will be converted to 1x1 matrices
      automatically.
398 # 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
399 vec3 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [-1, 2, 0, 0], [1, 3, 0, 0]]
400 # 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
401 vec2 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [1, 2, 0, 0], [-1, 3, 0, 0]]
402 # The first vector has five components as well but they are matrices of quads, not just
      the quads themselves.
403 m1 = [[[1, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
404 m2 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [1, 0, 1, 1]]]
405 m3 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
406 m4 = [[[0, 0, 0, 0], [0.5, 0, 0, 1]], [[0.5, 0, 0, 1], [0, 0, 0, 0]]]
407 m5 = [[[0, 1, 0, 0], [0.5, 1, 0, 1]], [[0.5, 1, 0, 1], [0, 1, 0, 0]]]
408 vec1 = [m1, m2, m3, m4, m5]
409
410 # The first rep must be the singlet even channel, where the unit operator resides.
411 # After this, the order doesn't matter.
412 # Spins for these again go even, even, odd.
413 # The Z2 even sector has only even spins, Z2 odd sector runs over even and odd spins.
414 info = [[vec1, 0, "z2-even-l-even"], [vec2, 0, "z2-odd-l-even"], [vec3, 1, "z2-odd-l-
      odd"]]
415
416 # We instantiate the SDP object, inputting our vectorial sum info.
417 # dim_list, convolved_block_table_list, vector_types (how they combine to compose sum
      rule).
418 # We use the first calculated SDP object as a prototype for all the rest.
419 # This is because some bounds remain unchanged, no need to recalculate basis.
420 # Basis is independent of external scaling dimensions, cares only of the bounds on
      particular operators.
421 if reference_sdp == None:
422     sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
423     reference_sdp = sdp
424 else:
425     sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info, protoype =
      reference_sdp)
426
427 # We assume the continuum in both Z2 odd / even sectors begins at the dimension=3.
428 sdp.set_bound([0, "z2-even-l-even"], self.dim)
429 sdp.set_bound([0, "z2-odd-l-even"], self.dim)
430
431 # Except for the two lowest dimension scalar operators in each sector.

```



```

432         sdp.add_point([0, "z2-even-l-even"], eps)
433         sdp.add_point([0, "z2-odd-l-even"], sig)
434
435         # We expect these calculations to be computationally intensive.
436         # We set maxThreads=16 to parallelise SDPB for all runs.
437         # See 'common.py' for the list of SDPB option strings, as well as their default values.
438         sdp.set_option("maxThreads", 16)
439         sdp.set_option("dualErrorThreshold", 1e-15)
440
441         # Run the SDP to determine if the current operator spectrum is permissable.
442         result = sdp.iterate()
443         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 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.table.append(grid)
460         self.save_grid(grid, self.name)
461
462     # A function used for the multi-correlator 3D Ising example.
463     # Note default is antisymmetrised convolved conformal blocks.
464     def convolved_table_list(self, tab1, tab2, tab3):
465         f_tab1a = bootstrap.ConvolvedBlockTable(tab1)
466         f_tab1s = bootstrap.ConvolvedBlockTable(tab1, symmetric = True)
467         f_tab2a = bootstrap.ConvolvedBlockTable(tab2)
468         f_tab2s = bootstrap.ConvolvedBlockTable(tab2, symmetric = True)
469         f_tab3 = bootstrap.ConvolvedBlockTable(tab3)
470         return [f_tab1a, f_tab1s, f_tab2a, f_tab2s, f_tab3]
471
472     '''
473     # We define a class with imposes a gap in the Z2-even operator sector.
474     # The continuum starts at a specified value, and we add an operator between this and unitarity
475     # bound.
476     class IsingGap(object):
477         bootstrap.cutoff=1e-10
478         def __init__(self, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults):
479             self.dim = dim
480             self.gap = gap
481             self.sig_values = sig_values
482             self.eps_values = eps_values
483             self.table = []
484             self.name = "name"
485             # if from_file == True:
486             # self.recover_table(file_name)
487             # else:
488             # self.table = []
489     '''

```