revised-ising-class.py

```python
# We create a 'master' Ising class, with options to gap the spectrum or use mixed correlator information.
import bootstrap
import matplotlib.pyplot as plt
import time, datetime
import datetime
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages

sig_defaults = np.arange(0.5,0.85,0.05).tolist()
eps_defaults = np.arange(1.0,2.2,0.2).tolist()

class Point(object):
  def __init__(self, sig, eps, kmax, lmax, mmax, nmax, allowed, run_time, cpu_time):
    self.sig = sig
    self.eps = eps
    self.kmax = kmax
    self.lmax = lmax
    self.mmax = mmax
    self.nmax = nmax
    self.allowed = allowed
    self.run_time = run_time
    self.cpu_time = cpu_time

class Grid(object):
  def __init__(self, kmax, lmax, mmax, nmax, allowed_points, disallowed_points, run_time, cpu_time):
    self.kmax = kmax
    self.lmax = lmax
    self.mmax = mmax
    self.nmax = nmax
    self.allowed_points = allowed_points
    self.disallowed_points = disallowed_points
    self.run_time = run_time
    self.cpu_time = cpu_time

class Ising(object):
  def __init__(self, name, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults):
    self.dim = dim
    self.gap = gap
    self.sig_values = sig_values
    self.eps_values = eps_values
    self.grid_table = []
    self.name = name

  # For a given set of conformal blocks, set by kmax and lmax, generate a grids for a specified range of mmax and nmax.
  # If we obtain a grid of entirely dissallowed points, fill in the rest of the grids for that kmax and lmax.
  def iterate_parameters(self, kmax_range, lmax_range, mmax_range, nmax_range):
    keys = self.generate_keys(kmax_range, lmax_range, mmax_range, nmax_range)

    while len(keys) > 0:
```

```
      # Used keys will store the keys for which there is already a grid in table.
      used_keys = []
      #null_keys = []

      for key in keys:
        if self.get_grid_index(key) != -1:
          used_keys.append(key)
          continue
        print("Trying kmax = " + str(key[0]) + ", lmax = " + str(key[1]) + ", mmax = " + str(key[2]) + ", nmax = " + str(key[3]))
        self.determine_grid(key)
        used_keys.append(key)

        # If the grid has only disallowed points...
        if self.grid_table[self.get_grid_index(key)].allowed_points == []:
          print ("In the if statement.")
          k = key[0]
          l = key[1]
          m = key[2]
          n = key[3]

          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]

          for key in null_keys:
            if self.get_grid_index(key) != -1:
              used_keys.append(key)
              continue
            #grid = Grid(*key, [], [])
            grid = Grid(*(key + [[], [], 0, 0]))

            for sig in self.sig_values:
              for eps in self.eps_values:
                grid.disallowed_points.append((sig, eps))

            self.grid_table.append(grid)
            self.save_grid(grid, self.name)

          break

      # We remove all keys from the list that we are done with.
      keys = [key for key in keys if key not in null_keys and key not in used_keys]
      null_keys = []


  '''
  # Saves the data as an executable file that will repopulate the table attribute.
  # Note, we now do this as we go, instead of at the end, to avoid loss of mass data.
  def save_to_file(self, name):
    with open(name + ".py", 'w') as file:
      file.write("self.table = []\n")
      for grid in self.table:
        file.write("kmax = " + str(grid.kmax) + "\n")
        file.write("lmax = " + str(grid.lmax) + "\n")
        file.write("mmax = " + str(grid.mmax) + "\n")
        file.write("nmax = " + str(grid.nmax) + "\n")
        file.write("allowed_points = " + str(grid.allowed_points) + "\n")
        file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")
```

```
        file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points, disallowed_points))" + "\n")
  '''

  def save_grid(self, grid, name):
    with open(name + ".py", 'a') as file:
      file.write("kmax = " + str(grid.kmax) + "\n")
      file.write("lmax = " + str(grid.lmax) + "\n")
      file.write("mmax = " + str(grid.mmax) + "\n")
      file.write("nmax = " + str(grid.nmax) + "\n")
      file.write("allowed_points = " + str(grid.allowed_points) + "\n")
      file.write("disallowed_points = " + str(grid.disallowed_points) + "\n")
      file.write("run_time = " + str(grid.run_time) + "\n")
      file.write("cpu_time = " + str(grid.cpu_time) + "\n")
      file.write("self.grid_table.append(Grid(kmax, lmax, mmax, nmax, allowed_points, disallowed_points, run_time, cpu_time))" + "\n")

  # Recoveres a table stored to a file.
  def recover_grid_table(self, file_name):
    exec(open(file_name + ".py").read())


  # Searches table of grids for index matching the input key. Returns -1 if not found.
  def get_grid_index(self, key):
    for i in range(0, len(self.grid_table)):
      if self.grid_table[i].kmax == key[0] and self.grid_table[i].lmax == key[1] and self.grid_table[i].mmax == key[2] and self.grid_table[i].nmax == key[3]:
        return i
    return -1

  # Plots a single grid, specified by a key. Note grid must be in grid_table.
  def plot_grid(self, key, name):
    grid = grid_table[self.get_grid_index(key)]
    allowed_sig = [points[0] for points in grid.allowed_points]
    allowed_eps = [points[1] for points in grid.allowed_points]
    disallowed_sig = [points[0] for points in grid.disallowed_points]
    disallowed_eps = [points[1] for points in grid.disallowed_points]

    # Plot a grid.
    plt.plot(allowed_sig, allowed_eps, 'r+')
    plt.plot(disallowed_sig, disallowed_eps, 'b+')
    plt.title('kmax : ' + grid.kmax.__str__() + " " +
        'lmax : ' + grid.lmax.__str__() + " " +
        'mmax : ' + grid.mmax.__str__() + " " +
        'nmax : ' + grid.nmax.__str__())

  # Plots and saves a series of grids to an output PDF file.
  # Takes as input parameter values for which we want plotted grids, and the desired PDF file name.
  def plot_grids(self, keys, file_name, plots_per_page, grid_size):
    table = self.generate_table(keys)
    pdf_pages = PdfPages(file_name + ".pdf")

    # Define the number of plots per page and the size of the grid board.
    nb_plots = len(grid_table)
    # nb_plots_per_page = 6
    nb_pages = int(np.ceil(nb_plots / float(plots_per_page)))
    # grid_size=(3,2)
```

```

    # This will define which row of the grid we are on.
    row_index = 0

    # We go through each 'grid' in 'grid_table', generating a plot for each.
    for i in range(nb_plots):
      # To begin, declare a new figure / page if we have exceeded limit of the last page.
      if i % plots_per_page == 0:
        fig = plt.figure(figsize=(8.27, 11.69), dpi=100)

      # Now, add a plot for the current grid on the grid board.
      plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
      if i % grid_size[1] == 1:
        row_index += 1

      # Handle our data. Retrieve isolated points for plotting from our input grid_table of Grid objects.
      allowed_sig = [points[0] for points in table[i].allowed_points]
      allowed_eps = [points[1] for points in table[i].allowed_points]
      disallowed_sig = [points[0] for points in table[i].disallowed_points]
      disallowed_eps = [points[1] for points in table[i].disallowed_points]

      # Plot a grid.
      plt.plot(allowed_sig, allowed_eps, 'r+')
      plt.plot(disallowed_sig, disallowed_eps, 'b+')
      plt.title('kmax : ' + table[i].kmax.__str__() + " " +
          'lmax : ' + table[i].lmax.__str__() + " " +
          'mmax : ' + table[i].mmax.__str__() + " " +
          'nmax : ' + table[i].nmax.__str__())

      # If we have filled a page, or have reached the end of our plots, tight-pack and save the page.
      if (i + 1) % plots_per_page == 0 or (i + 1) == nb_plots:
        plt.tight_layout()
        pdf_pages.savefig(fig)
        row_index = 0

    pdf_pages.close()

  # Returns a key or list of keys generated by the input parameter ranges.
  def generate_keys(self, kmax_range, lmax_range, mmax_range, nmax_range):
    if type(kmax_range) == int:
      kmax_range = [kmax_range]
    if type(lmax_range) == int:
      lmax_range = [lmax_range]
    if type(mmax_range) == int:
      mmax_range = [mmax_range]
    if type(nmax_range) == int:
      nmax_range = [nmax_range]
    keys = []
    for kmax in kmax_range:
      for lmax in lmax_range:
        for mmax in mmax_range:
          for nmax in nmax_range:
            key = [kmax, lmax, mmax, nmax]
            keys.append(key)
    return keys

```

```
# Generates a subtable table of desired, already determined grids from main table.
# Gives a warning message if a grid isn't found.
def generate_table(self, keys):
  # table to store the resulting grids.
  table = []
  for key in keys:
    if self.get_grid_index(key) == -1:
      print("Grid at kmax = " + str(key[0]) + ", " +
        "lmax = " + str(key[1]) + ", " +
        "mmax = " + str(key[2]) + ", " +
        "nmax = " + str(key[3]) + ", " + "does not exist.")
    else:
      table.append(self.table[self.get_grid_index(key)])

  return table

# Takes two keys and returns a dictionary with the direction of every point.
def changes(self, key1, key2):
  changes = {}
  allowed_one = self.grid_table[self.get_grid_index(key1)].allowed_points
  allowed_two = self.grid_table[self.get_grid_index(key2)].allowed_points

  for sig in self.sig_values:
    for eps in self.eps_values:
      if (sig, eps) in allowed_one and (sig, eps) in allowed_two:
        changes[(sig, eps)] = 0
      if (sig, eps) not in allowed_one and (sig, eps) not in allowed_two:
        changes[(sig, eps)] = 0
      if (sig, eps) in allowed_one and (sig, eps) not in allowed_two:
        changes[(sig, eps)] = -1
      if (sig, eps) not in allowed_one and (sig, eps) in allowed_two:
        changes[(sig, eps)] = 1
  return changes

# grid_size is a tuple of (rows, columns).
def plot_changes(self, keys, file_name, plots_per_page, grid_size):
  pdf_pages = PdfPages(file_name + ".pdf")

  # Define the number of plots per page and the size of the grid board.
  # We have one less plots than grids.
  nb_plots = len(keys)
  # nb_plots_per_page = 6
  nb_pages = int(np.ceil(nb_plots / float(plots_per_page)))
  # grid_size=(3,2)

  # This will define which row of the grid we are on.
  row_index = 0

  # We go through each 'grid' in 'grid_table', generating a plot for each.
  for i in range(nb_plots):
    # To begin, declare a new figure / page if we have exceeded limit of the last page.
    # 8.27 x 11.69 dimensions of A4 page in inches. DPI - dots per inch (resolution.)
    if i % plots_per_page == 0:
      fig = plt.figure(figsize=(8.27, 11.69), dpi=100)

    # Now, add a plot for the current grid on the grid board.
    plt.subplot2grid(grid_size, (row_index, i % grid_size[1]))
    if i % grid_size[1] == 1:
```

```
      row_index += 1

    # We want the first grid to compare all changes to.
    if i == 0:
      grid = self.grid_table[self.get_grid_index(keys[i])]
      allowed_sig = [points[0] for points in grid.allowed_points]
      allowed_eps = [points[1] for points in grid.allowed_points]
      disallowed_sig = [points[0] for points in grid.disallowed_points]
      disallowed_eps = [points[1] for points in grid.disallowed_points]

      # Plot the grid.
      plt.plot(allowed_sig, allowed_eps, 'r+')
      plt.plot(disallowed_sig, disallowed_eps, 'b+')
      plt.title('kmax : ' + grid.kmax.__str__() + " " +
          'lmax : ' + grid.lmax.__str__() + " " +
          'mmax : ' + grid.mmax.__str__() + " " +
          'nmax : ' + grid.nmax.__str__())

      y_range = plt.ylim()
      x_range = plt.xlim()

    else:
      changes = self.changes(keys[i-1], keys[i])
      unchanged_points = []
      to_allowed_points = []
      to_disallowed_points = []
      for point in changes:
        if changes[point] == 0:
          unchanged_points.append(point)
        if changes[point] == 1:
          to_allowed_points.append(point)
        if changes[point] == -1:
          to_disallowed_points.append(point)

      unchanged_sig = [points[0] for points in unchanged_points]
      unchanged_eps = [points[1] for points in unchanged_points]
      to_disallowed_sig = [points[0] for points in to_disallowed_points]
      to_disallowed_eps = [points[1] for points in to_disallowed_points]
      to_allowed_sig = [points[0] for points in to_allowed_points]
      to_allowed_eps = [points[1] for points in to_allowed_points]

      # Plot a grid.
      plt.plot(to_allowed_sig, to_allowed_eps, 'r+')
      plt.plot(to_disallowed_sig, to_disallowed_eps, 'b+')
      plt.xlim(x_range)
      plt.ylim(y_range)
      plt.title('kmax : ' + self.grid_table[self.get_grid_index(keys[i])].kmax.__str__() + " " +
          'lmax : ' + self.grid_table[self.get_grid_index(keys[i])].lmax.__str__() + " " +
          'mmax : ' + self.grid_table[self.get_grid_index(keys[i])].mmax.__str__() + " " +
          'nmax : ' + self.grid_table[self.get_grid_index(keys[i])].nmax.__str__())

    # If we have filled a page, or have reached the end of our plots, tight-pack and save the page.
    if (i + 1) % plots_per_page == 0 or (i + 1) == nb_plots:
      plt.tight_layout()
      pdf_pages.savefig(fig)
      row_index = 0
```

```

    pdf_pages.close()

class SingleCorrelator(Ising):
  bootstrap.cutoff=1e-10
  def __init__(self, name, dim = 3, gap = 3, sig_values = sig_defaults, eps_values = eps_defaults):
    self.dim = dim
    self.gap = gap
    self.sig_values = sig_values
    self.eps_values = eps_values
    self.grid_table = []
    self.name = name

  # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
  def determine_grid(self, key):
    #if self.get_grid_index(key) != -1:
    start_time=time.time()
    start_cpu=time.clock()
    tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
    tab2 = bootstrap.ConvolvedBlockTable(tab1)

    # Instantiate a Grid object with appropriate input values.
    # grid=Grid(*key, [], [])
    grid = Grid(*(key + [[], [], 0, 0]))

    for sig in self.sig_values:
      for eps in self.eps_values:
        sdp = bootstrap.SDP(sig, tab2)
        # SDPB will naturally try to parallelize across 4 cores / slots.
        # To prevent this, we set its 'maxThreads' option to 1.
        # See 'common.py' for the list of SDPB option strings, as well as their default values.
        sdp.set_option("maxThreads", 1)
        sdp.set_bound(0, float(self.gap))
        sdp.add_point(0, eps)
        result = sdp.iterate()
        if result:
          grid.allowed_points.append((sig, eps))
        else:
          grid.disallowed_points.append((sig, eps))

    # Now append this grid object to the IsingGap grid_table.
    # Note we will need to implement a look up table to retrieve desired data.
    end_time=time.time()
    end_cpu=time.clock()
    run_time=end_time-start_time
    cpu_time=end_cpu-start_cpu
    run_time = datetime.timedelta(seconds = int(end_time - start_time))
    cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))

    grid.run_time = run_time
    grid.cpu_time = cpu_time
    self.grid_table.append(grid)
    self.save_grid(grid, self.name)

# For mixed correlator, we pass pairs of external scaling dimensions to the SDP.
# We copy the content of the triples entering the SDP from the tutorial, same case.
# We want to scan over all possible [sig, eps], assuming only one relevant Z2-even and Z2-odd
```

```
    operator.
# Use a protoype to use the same basis for all SDPs, so we don't need to recaculate bases.
# Dump the ConformalBlockTable objects once we have used them to save memory.
# Set dualThresholdError to 1e-15.
# Use 16 cores for all SDP runs - set maxThreads = 16, speed up the SDP.
class MixedCorrelator(Ising):
  sig_range_def = [0.5179, 0.5186]
  eps_range_def = [1.4100, 1.4150]
  sig_step_def = 0.000005
  eps_step_def = 0.00005
  bootstrap.cutoff=0
  def __init__(self, name, dim = 3, sig_range = sig_range_def, eps_range = eps_range_def, sig_step = sig_step_def, eps_step = eps_step_def):
    self.name = name
    self.dim = dim
    self.sig_range = sig_range
    self.eps_range = eps_range
    self.sig_step = sig_step
    self.eps_step = eps_step
    self.point_table = []
    self.grid_table = []

  # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
  def determine_points(self, key):
    # key = [self.kmax, self.lmax, self.mmax, self.nmax]
    #if self.get_grid_index(key) != -1:
    reference_sdp = None

    # Constant sig-eps lines: eps = sig - c.
    # Choose a starting point for each line. (0.5179, 1.4110).
    # g_tab1 and g_tab3 don't change on a given line of constant delta{sig,eps}.
    sig_values = np.arange(self.sig_range[0], self.sig_range[1] + self.sig_step, self.sig_step).tolist()
    for eps in np.arange(self.eps_range[0], self.eps_range[1] + self.eps_step, self.eps_step).tolist():
      # sig_values = []
      eps_values = []
      # For each value of x along this line:
      for sig in sig_values:
        # sig_values.append(sig)
        eps_values.append(eps + (sig-self.sig_range[0]))

      # Could initiate all blocks prior to loop here using sig_values[0].
      # However, want to capture the timing of this within the first point?
      blocks_initiated = False
      for i in range(len(sig_values)):
        sig = sig_values[i]
        eps = eps_values[i]
        start_time=time.time()
        start_cpu=time.clock()
        # Generate three conformal block tables, two of which depend on the dimension differences.
        # They need only be calculated once for any given diagonal. They remain constant along this line.
        # Uses the function above to return the 5 ConvolvedConformalBlocks we need.
        # The ConvolvedConformalBlock objects inherits the dimension differences from ConformalBlockTable.
        # We set odd_spins = True for odd those ConvolvedConformalBlocks appearing in odd-sector-
```

```
    odd-spins.
    # We set symmetric = True where required.
    if blocks_initiated == False:
      g_tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
      g_tab2 = bootstrap.ConformalBlockTable(self.dim, *(key + [eps-sig, sig-eps, "odd_spins = True"]))
      g_tab3 = bootstrap.ConformalBlockTable(self.dim, *(key + [sig-eps, sig-eps, "odd_spins = True"]))
      tab_list = self.convolved_table_list(g_tab1, g_tab2, g_tab3)
      for tab in [g_tab1, g_tab2, g_tab3]:
        tab.dump("tab_" + str(tab.delta_12) + "_" + str(tab.delta_34))
        del tab
      blocks_initiated = True

    # N.B vec3 & vec2 are 'raw' quads, which will be converted to 1x1 matrices automatically.
    # 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
    vec3 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [-1, 2, 0, 0], [1, 3, 0, 0]]
    # 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
    vec2 = [[0, 0, 0, 0], [0, 0, 0, 0], [1, 4, 1, 0], [1, 2, 0, 0], [-1, 3, 0, 0]]
    # The first vector has five components as well but they are matrices of quads, not just the quads themselves.
    m1 = [[[1, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
    m2 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [1, 0, 1, 1]]]
    m3 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
    m4 = [[[0, 0, 0, 0], [0.5, 0, 0, 1]], [[0.5, 0, 0, 1], [0, 0, 0, 0]]]
    m5 = [[[0, 1, 0, 0], [0.5, 1, 0, 1]], [[0.5, 1, 0, 1], [0, 1, 0, 0]]]
    vec1 = [m1, m2, m3, m4, m5]

    # The first rep must be the singlet even channel, where the unit operator resides.
    # After this, the order doesn't matter.
    # Spins for these again go even, even, odd.
    # The Z2 even sector has only even spins, Z2 odd sector runs over even and odd spins.
    info = [[vec1, 0, "z2-even-l-even"], [vec2, 0, "z2-odd-l-even"], [vec3, 1, "z2-odd-l-odd"]]

    # We instantiate the SDP object, inputting our vectorial sum info.
    # dim_list, convolved_block_table_list, vector_types (how they combine to compose sum rule).
    # We use the first calculated SDP object as a prototype for all the rest.
    # This is because some bounds remain unchanged, no need to recalculate basis.
    # Basis is independent of external scaling dimensions, cares only of the bounds on particular operators.
    sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
    if reference_sdp == None:
      sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info)
      reference_sdp = sdp
    else:
      sdp = bootstrap.SDP([sig, eps], tab_list, vector_types = info, prototype = reference_sdp)

    # We assume the continuum in both Z2 odd / even sectors begins at the dimension=3.
    sdp.set_bound([0, "z2-even-l-even"], self.dim)
    sdp.set_bound([0, "z2-odd-l-even"], self.dim)

    # Except for the two lowest dimension scalar operators in each sector.
    sdp.add_point([0, "z2-even-l-even"], eps)
```

```
        sdp.add_point([0, "z2-odd-l-even"], sig)

        # We expect these calculations to be computationally intensive.
        # We set maxThreads=16 to parallelise SDPB for all runs.
        # See 'common.py' for the list of SDPB option strings, as well as their default values.
        sdp.set_option("maxThreads", 16)
        sdp.set_option("dualErrorThreshold", 1e-15)

        # Run the SDP to determine if the current operator spectrum is permissable.
        print("Testing point " + "(" + sig.__str__() + ", " + eps.__str__() +")...")
        result = sdp.iterate()
        end_time = time.time()
        end_cpu = time.clock()
        run_time = datetime.timedelta(seconds = int(end_time - start_time))
        cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))

        point = Point(*([sig, eps] + key + [result, run_time, cpu_time]))
        self.point_table.append(point)
        self.save_point(point, self.name)

  # Determines a full grid of Points.
  # Appends the Points to point_table and the Grid to grid_table.
  def determine_grid(self, key):
    #if self.get_grid_index(key) != -1:
    #start_time=time.time()
    #start_cpu=time.clock()

    grid = Grid(*(key + [[], [], 0, 0]))

    self.determine_points(key)

    # end_time=time.time()
    # end_cpu=time.clock()
    # run_time = datetime.timedelta(seconds = int(end_time - start_time))
    # cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))

    points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax,
        points.nmax] == key]
    for point in points:
      grid.run_time += point.run_time
      grid.cpu_time += point.cpu_time
      if point.allowed == True:
        grid.allowed_points.append((point.sig, point.eps))
      else:
        grid.disallowed_points.append((point.sig, point.eps))

    # grid.run_time = run_time
    # grid.cpu_time = cpu_time
    self.grid_table.append(grid)
    # self.save_grid(grid, self.name)

  # A method for composing a whole grid from a set of 'raw' points.
  # Allows more flexability - can choose sets of disparate points or use parallelization.
  def make_grid(self, key):
    grid = Grid(*(key + [[], [], 0, 0]))
    points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax,
        points.nmax] == key]
    for point in points:
```

```
      grid.run_time += point.run_time
      grid.cpu_time += point.cpu_time
      if point.allowed == True:
        grid.allowed_points.append((point.sig, point.eps))
      else:
        grid.disallowed_points.append((point.sig, point.eps))

  # A function used for the multi-correlator 3D Ising example.
  # Note default is antisymmetrised convolved conformal blocks.
  def convolved_table_list(self, tab1, tab2, tab3):
    f_tab1a = bootstrap.ConvolvedBlockTable(tab1)
    f_tab1s = bootstrap.ConvolvedBlockTable(tab1, symmetric = True)
    f_tab2a = bootstrap.ConvolvedBlockTable(tab2)
    f_tab2s = bootstrap.ConvolvedBlockTable(tab2, symmetric = True)
    f_tab3 = bootstrap.ConvolvedBlockTable(tab3)
    return [f_tab1a, f_tab1s, f_tab2a, f_tab2s, f_tab3]

  # Returns the number of points that will be calculated for given sig,eps ranges and step sizes.
  def points(self):
    return ((self.sig_range[1] - self.sig_range[0])/self.sig_step) * ((self.eps_range[1] - self.eps_range[0])/self.eps_step)

  # Saves a Point object' data to file named in self.name
  def save_point(self, point, name):
    with open(name + ".py", 'a') as file:
      file.write("kmax = " + str(point.kmax) + "\n")
      file.write("lmax = " + str(point.lmax) + "\n")
      file.write("mmax = " + str(point.mmax) + "\n")
      file.write("nmax = " + str(point.nmax) + "\n")
      file.write("sig = " + str(point.sig) + "\n")
      file.write("eps = " + str(point.eps) + "\n")
      file.write("allowed = " + str(point.allowed) + "\n")
      file.write("run_time = " + str(point.run_time) + "\n")
      file.write("cpu_time = " + str(point.cpu_time) + "\n")
      file.write("self.point_table.append(Point(kmax, lmax, mmax, nmax, sig, eps, run_time, cpu_time))" + "\n")

  # Recoveres a table of Point objects stored to a file.
  def recover_points(self, file_name):
    exec(open(file_name + ".py").read())
```