revised-ising-gap.py

```python
import bootstrap
import matplotlib.pyplot as plt
import time
import datetime
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages

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

# We define a class with imposes a gap in the Z_2-even operator sector.
# The continuum starts at a specified value, and we add an operator between this and unitarity
    bound.
class IsingGap(object):
  bootstrap.cutoff=1e-10
  def __init__(self, from_file = False, file_name = 'name', dim = 3, gap = 3, sig_values = np.
      arange(0.5,0.85,0.05).tolist(), eps_values = np.arange(1.0,2.2,0.2).tolist()):
    self.dim = dim
    self.gap = gap
    self.sig_values = sig_values
    self.eps_values = eps_values
    if from_file == True:
      self.recover_table(file_name)
    else:
      self.table = []

  # Determines allowed and disallowed scaling dimensions for whatever the parameters are.
  def determine_grid(self, key):
    tab1 = bootstrap.ConformalBlockTable(self.dim, *key)
    tab2 = bootstrap.ConvolvedBlockTable(tab1)

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

    for sig in self.sig_values:
      for eps in self.eps_values:
        sdp = bootstrap.SDP(sig,tab2)
        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 table.
    # Note we will need to implement a look up table to retrieve desired data.
    self.table.append(grid)
```

```

  # Append to the table more grids specified by parameter and parameter range.
  def iterate_parameters(self, kmax_range, lmax_range, mmax_range, nmax_range):
    keys = self.generate_keys(kmax_range, lmax_range, mmax_range, nmax_range)
    for key in keys:
      if self.get_grid_index(key) != -1:
        continue
      self.determine_grid(key)

  # Saves the data as an executable file that will repopulate the table attribute.
  def save_to_file(self, name):
    with open(name + ".py", 'a') 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.table.append(Grid(kmax, lmax, mmax, nmax, allowed_points, disallowed_points))" + "\n")

  # Recoveres a table stored to a file.
  def recover_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.table)):
      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]:
        return i
    return -1

  # 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):
    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(table)
    nb_plots_per_page = 6
    nb_pages = int(np.ceil(nb_plots / float(nb_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 '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 % nb_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 out input 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) % nb_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

  def convergence_factor(self, key):
    grid = self.table[self.get_grid_index(key)]
    #key = self.generate_keys(grid.kmax, grid.lmax, grid.mmax, grid.nmax)[0]
    grid_value = abs(len(grid.allowed_points) - len(grid.disallowed_points))

    convergence = 0
    for i in range(len(key)):
      key[i] += 1
      if self.get_grid_index(key) == -1:
        print ("Can’t calculate convergence factor. The required grids have not been calculated."
            )
        break
      else:
        next_grid = self.table[self.get_grid_index(key)]
        next_grid_value = abs(len(next_grid.allowed_points) - len(next_grid.disallowed_points))
        convergence += abs(grid_value - next_grid_value)
      key[i] -= 1
    convergence /= len(key)

    return convergence
```