archipelago.py

```python
import subprocess
import bootstrap
import matplotlib.pyplot as plt
import time, datetime
import datetime
import numpy as np
from matplotlib.backends.backend_pdf import PdfPages
from symengine.lib.symengine_wrapper import *

class MixedCorrelator(object):
  bootstrap.cutoff=0
  def __init__(self, N, dim = 3):
    self.dim = dim
    self.N = N
    self.grid_table = []
    self.point_table = []
    self.grid_file = "grid_saves"
    self.point_file = "point_saves"

    # Insert vector info here.
    # Must be associated with a particular instance of the MixedCorrelator object, since the vec info is N-dependent.
    v2 = [[1, 0, 0, 0], [(1 - 2/self.N), 0, 0, 0], [-(1 + 2/self.N), 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0]]
    v3 = [[1, 0, 0, 0], [-1, 0, 0, 0], [1, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0]]
    v4 = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [1, 2, 1, 0], [1, 3, 0, 0], [-1, 4, 0, 0]]
    v5 = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], [1, 2, 1, 0], [-1, 3, 0, 0], [1, 4, 0, 0]]
    m1 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
    m2 = [[[1, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
    m3 = [[[1, 1, 0, 0], [0, 1, 0, 0]], [[0, 1, 0, 0], [0, 1, 0, 0]]]
    m4 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [1, 0, 1, 1]]]
    m5 = [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
    m6 = [[[0, 0, 0, 0], [0.5, 0, 0, 1]], [[0.5, 0, 0, 1], [0, 0, 0, 0]]]
    m7 = [[[0, 1, 0, 0], [0.5, 1, 0, 1]], [[0.5, 1, 0, 1], [0, 1, 0, 0]]]
    v1 = [m1, m2, m3, m4, m5, m6, m7]
    self.info = [[v1, 0, 0], [v2, 0, 1], [v3, 1, 2], [v4, 0, 3], [v5, 1, 4]]

  def determine_grid(self, key, row_lists):
    grid = Grid(*(key + [[], [], 0, 0]))
    for row in row_lists:
      self.determine_row(key, row)

    points = [point for point in self.point_table if [point.kmax, point.lmax, point.mmax, point.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))

    self.grid_table.append(grid)
    grid.save(self.grid_file)

  def determine_row(self, key, row):
  # Will be called with a given row_lists[i]
  # Use generate_rows() method to build row_lists.
    # row = row_lists[row_index]
    reference_sdp = None
    blocks_initiated = False
    for i in range(len(row[0])):
      phi = eval_mpfr(row[0][i], bootstrap.prec)
      sing = eval_mpfr(row[1][i], bootstrap.prec)

      # phi_sing = eval_mpfr(phi - sing, bootstrap.prec)
      # sing_phi = eval_mpfr(sing - phi, bootstrap.prec)

      start = time.time()
      start_cpu = time.clock()

      if blocks_initiated == False:
        g_tab1 = bootstrap.ConformalBlockTable(self.dim, *(key + [0, 0, "odd_spins = True"]))
        g_tab2 = bootstrap.ConformalBlockTable(self.dim, *(key + [phi - sing, phi - sing, "odd_spins = True"]))
        g_tab3 = bootstrap.ConformalBlockTable(self.dim, *(key + [sing - phi, phi - sing, "odd_spins = True"]))

        f_tab1a = bootstrap.ConvolvedBlockTable(g_tab1)
        f_tab1s = bootstrap.ConvolvedBlockTable(g_tab1, symmetric = True)
        f_tab2a = bootstrap.ConvolvedBlockTable(g_tab2)
        f_tab3a = bootstrap.ConvolvedBlockTable(g_tab3)
        f_tab3s = bootstrap.ConvolvedBlockTable(g_tab3, symmetric = True)

        tab_list = [f_tab1a, f_tab1s, f_tab2a, f_tab3a, f_tab3s]

        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

      max_dimension = 0
      for tab in tab_list:
        max_dimension = max(max_dimension, len(tab.table[0].vector))

      print("kmax should be around" + max_dimension.__str__() + ".")
      dimension = (5 * len(f_tab1a.table[0].vector)) + (2 * len(f_tab1s.table[0].vector))
      bootstrap.cb_end = time.time()
      bootstrap.cb_end_cpu = time.clock()
      cb_time = datetime.timedelta(seconds = int(bootstrap.cb_end - start))
      cb_cpu = datetime.timedelta(seconds = int(bootstrap.cb_end_cpu - start_cpu))
      print("The calculation of the required conformal blocks has successfully completed.")
      print("Time taken: " + str(cb_time))
      print("CPU_time: " + str(cb_cpu))

      if reference_sdp == None:
        sdp = bootstrap.SDP([phi, sing], tab_list, vector_types = self.info)
        reference_sdp = sdp
      else:
        sdp = bootstrap.SDP([phi, sing], tab_list, vector_types = self.info, prototype = reference_sdp)

      # We assume the continuum in both even vector and even singlet sectors begins at the dimension=3.
      sdp.set_bound([0, 0], self.dim)
      sdp.set_bound([0, 3], self.dim)

      # Except for the two lowest dimension scalar operators in each sector.
      sdp.add_point([0, 0], sing)
      sdp.add_point([0, 3], phi)


      sdp.set_option("maxThreads", 16)
      sdp.set_option("dualErrorThreshold", 1e-15)
      sdp.set_option("maxIterations", 1000)

      # Run the SDP to determine if the current operator spectrum is permissable.
      print("Testing point " + "(" + phi.__str__() + ", " + sing.__str__() + "with" + dimension.__str__() + "components.")
      result = sdp.iterate()
      end = time.time()
      end_cpu = time.clock()
      sdp_time = datetime.timedelta(seconds = int(end - bootstrap.xml_end))
      sdp_cpu = datetime.timedelta(seconds = int(end_cpu - bootstrap.xml_end_cpu))
      run_time = datetime.timedelta(seconds = int(end - start))
      cpu_time = datetime.timedelta(seconds = int(end_cpu - start_cpu))

      print("The SDP has finished running.")
      print("Time taken: " + str(sdp_time))
      print("CPU_time: " + str(sdp_cpu))
      print("See point file for more information. Check the times are consistent")

      point = Point(*([phi, sing] + key + [components, max_dimension, result, run_time, cpu_time, cb_time, cb_cpu, bootstrap.xml_time, bootstrap.xml_cpu, sdp_time, sdp_cpu]))
      self.point_table.append(point)
      point.save(self.point_file)

  # 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 + [components, [], [], 0, 0]))
    points = [points for points in self.point_table if [points.kmax, points.lmax, points.mmax, points.nmax] == key]
    # Points with the same key will have the same number of components.
    grid.components = point[0].components
    grid.max_dimension = point[0].max_dimension
    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.phi, point.sing))
      else:
        grid.disallowed_points.append((point.phi, point.sing))

  def load_table(self, file_name):
    #exec(open(file_name + ".py").read())
    with open(file_name + ".py") as infile:
      for line in infile:
        exec(line)


  # 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):
    grid = grid_table[self.get_grid_index(key)]
    allowed_phi = [points[0] for points in grid.allowed_points]
    allowed_sing = [points[1] for points in grid.allowed_points]
    disallowed_phi = [points[0] for points in grid.disallowed_points]
    disallowed_sing = [points[1] for points in grid.disallowed_points]

    # Plot a grid.
    plt.plot(allowed_phi, allowed_sing, 'r+')
    plt.plot(disallowed_phi, disallowed_sing, '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):
    tab = self.generate_table(keys)
    table = [grid for grid in tab if grid.run_time != 0]
    #table = self.grid_table
    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(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_phi = [points[0] for points in table[i].allowed_points]
      allowed_sing = [points[1] for points in table[i].allowed_points]
      disallowed_phi = [points[0] for points in table[i].disallowed_points]
      disallowed_sing = [points[1] for points in table[i].disallowed_points]

      # Plot a grid.
      plt.plot(allowed_phi, allowed_sing, ’r+’)
      plt.plot(disallowed_phi, disallowed_sing, ’b+’)
      plt.title(’[’ + table[i].kmax.__str__() + ", "
            + table[i].lmax.__str__() + ", "
            + table[i].mmax.__str__() + ", "
            + table[i].nmax.__str__() + ": "
            + table[i].components.__str__() + ", "
            + table[i].max_dimension.__str__() + ’]’
            + "     " + time.strftime(’%H:%M:%S’, table[i].run_time))

      # 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()

  def generate_table(self, keys):
    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.grid_table[self.get_grid_index(key)])

    return table

  def generate_rows(self, start, stop, phi_num, sing_num):
    # Generate grid of points and row_lists, to index in determine_points
    # phi_step = 0.0005
    # sing_step = 0.005

    # v1 = [0, sing_step]
    # v2 = [phi_step, phi_step]

    # start = [0.516, 1.39]
    # stop = [0.523, 1.44]

    # phi_range = np.linspace(start[0], stop[0], num=(stop[0]-start[0])/phi_step, endpoint=True, retstep=False, dtype=None).tolist()
    # sing_range = np.linspace(start[1], stop[1], num=(stop[1]-start[1])/sing_step, endpoint=True, retstep=False, dtype=None).tolist()
    phi_range = np.linspace(start[0], stop[0], num=phi_num, endpoint=True, retstep=False, dtype=None).tolist()
    sing_range = np.linspace(start[1], stop[1], num=sing_num, endpoint=True, retstep=False, dtype=None).tolist()

    phi_start = start[0]
    sing_start = start[1]

    phi_values = []
    sing_values = []

    row_lists = []
    phis = []
    sings = []
    for r in range(len(sing_range)):
      sing_row_start = sing_range[r]
      phi_row_start = phi_range[0]
      row_lists.append([])
      for s in range(len(phi_range)):
        phis.append(phi_range[s])
        sings.append(sing_row_start + (phi_range[s] - phi_row_start))
      row_lists[r].append(phis)
      row_lists[r].append(sings)
      phis = []
      sings = []

    return row_lists

class Point(object):
  def __init__(self, phi, sing, kmax, lmax, mmax, nmax, components, max_dimension, allowed,
      run_time, cpu_time, cb_time, cb_cpu, xml_time, xml_cpu, sdp_time, sdp_cpu):
    self.phi = phi
    self.sing = sing
    self.kmax = kmax
    self.lmax = lmax
    self.mmax = mmax
    self.nmax = nmax
    self.components = components
    self.max_dimension = max_dimension
    self.allowed = allowed
    self.run_time = run_time
    self.cpu_time = cpu_time
    self.cb_time = cb_time
    self.cb_cpu = cb_cpu
    self.xml_time = xml_time
    self.xml_cpu = xml_cpu
    self.sdp_time = sdp_time
    self.sdp_cpu = sdp_cpu

  # Saves a Point object’ data to file named in self.name
  def save(self, name):
    with open(name + ".py", ’a’) as file:
      file.write("phi = " + str(self.phi) + "\n")
      file.write("sing = " + str(self.sing) + "\n")
      file.write("kmax = " + str(self.kmax) + "\n")
      file.write("lmax = " + str(self.lmax) + "\n")
      file.write("mmax = " + str(self.mmax) + "\n")
      file.write("nmax = " + str(self.nmax) + "\n")
      file.write("components = " + str(self.components) + "\n")
      file.write("max_dimension = " + str(self.max_dimension) + "\n")
      file.write("allowed = " + str(self.allowed) + "\n")
      file.write("run_time = " + str(self.run_time) + "\n")
      file.write("cpu_time = " + str(self.cpu_time) + "\n")
      file.write("cb_time = " + str(self.cb_time) + "\n")
      file.write("cb_cpu = " + str(self.cb_cpu) + "\n")
      file.write("xml_time = " + str(self.xml_time) + "\n")
      file.write("xml_cpu = " + str(self.xml_cpu) + "\n")
      file.write("sdp_time = " + str(self.sdp_time) + "\n")
      file.write("sdp_cpu = " + str(self.sdp_cpu) + "\n")
      file.write("self.point_table.append(Point(phi, sing, kmax, lmax, mmax, nmax, components, max_dimension, allowed, run_time, cpu_time, cb_time, cb_cpu, xml_time, xml_cpu, sdp_time, sdp_cpu))" + "\n")

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

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