```
from __future__ import annotations

import numpy as np

from ase.units import Ha

from gpaw.pw.descriptor import PWMapping

from gpaw.response.pw_parallelization import (Blocks1D,
                                              PlaneWaveBlockDistributor)
from gpaw.response.frequencies import (FrequencyDescriptor,
                                       ComplexFrequencyDescriptor)
from gpaw.response.pair_functions import (SingleQPWDescriptor,
                                          map_ZgG_array_to_reduced_pd)


class Chi0RelatedData:
    """Base class for chi0 related data objects.

    Right now, all we do is to limit boiler plate code..."""

    def __init__(self,
                 wd: FrequencyDescriptor,
                 qpd: SingleQPWDescriptor):
        self.wd = wd
        self.qpd = qpd
        self.q_c = qpd.q_c

        # Basis set size
        self.nG = qpd.ngmax
        self.nw = len(wd)


class Chi0BodyData(Chi0RelatedData):
    """Data object containing the response body data arrays
    for a single q-point, while holding also the corresponding
    basis descriptors and block distributor."""

    def __init__(self, wd, qpd,
                 blockdist: PlaneWaveBlockDistributor):
        super().__init__(wd, qpd)

        # Initialize block distibution of plane wave basis
        self.blockdist = blockdist
        self.blocks1d = Blocks1D(blockdist.blockcomm, self.nG)

        # Data array
        self.data_WgG = self.zeros()

    def zeros(self):
        return np.zeros(self.WgG_shape, complex)

    @property
    def mynG(self):
        return self.blocks1d.nlocal

    @property
    def WgG_shape(self):
        return (self.nw, self.mynG, self.nG)

    def get_distributed_frequencies_array(self):
        """Copy data to a 'wGG'-like array, distributed over the entire world.

        This differs from copy_array_with_distribution('wGG'), in that the
        frequencies are distributed over world, instead of among the block
        communicator."""
        return self.blockdist.distribute_frequencies(self.data_WgG, self.nw)

    def copy_array_with_distribution(self, distribution):
        """Copy data to a new array of a desired distribution.

        Parameters
        ----------
        distribution: str
            Array distribution. Choices: 'wGG' and 'WgG'
        """
        data_x = self.blockdist.distribute_as(self.data_WgG, self.nw,
                                              distribution)

        if data_x is self.data_WgG:
            # When asking for 'WgG' distribution or when there is no block
            # distribution at all, we may still be pointing to the original
            # array, but we want strictly to return a copy
            assert distribution == 'WgG' or \
                self.blockdist.blockcomm.size == 1
            data_x = self.data_WgG.copy()

        return data_x

    def copy_with_reduced_pd(self, qpd):
        """Make a copy corresponding to a new plane-wave description."""
        new_chi0_body = Chi0BodyData(self.wd, qpd, self.blockdist)

        # Map data to reduced plane-wave representation
        new_chi0_body.data_WgG[:] = map_ZgG_array_to_reduced_pd(
            self.qpd, qpd, self.blockdist, self.data_WgG)

        return new_chi0_body


class Chi0DrudeData:
    def __init__(self, zd: ComplexFrequencyDescriptor):
        self.zd = zd
        self.plasmafreq_vv, self.chi_Zvv = self.zeros()

    def zeros(self):
        return (np.zeros(self.vv_shape, complex),  # plasmafreq
                np.zeros(self.Zvv_shape, complex))  # chi0_drude

    @staticmethod
    def from_frequency_descriptor(wd, rate):
        """Construct the Chi0DrudeData object from a frequency descriptor and
        the imaginary part (in eV) of the resulting horizontal frequency
        contour"""
        rate = rate / Ha  # eV -> Hartree
        zd = ComplexFrequencyDescriptor(wd.omega_w + 1.j * rate)

        return Chi0DrudeData(zd)

    @property
    def nz(self):
        return len(self.zd)

    @property
    def vv_shape(self):
        return (3, 3)

    @property
    def Zvv_shape(self):
        return (self.nz,) + self.vv_shape


class Chi0OpticalExtensionData(Chi0RelatedData):
    def __init__(self, wd, qpd):
        assert qpd.optical_limit
        super().__init__(wd, qpd)

        self.head_Wvv, self.wings_WxvG = self.zeros()
    def zeros(self):
        return (np.zeros(self.Wvv_shape, complex),  # head
                np.zeros(self.WxvG_shape, complex))  # wings

    @property
    def Wvv_shape(self):
        return (self.nw, 3, 3)

    @property
    def WxvG_shape(self):
        return (self.nw, 2, 3, self.nG)

    def copy_with_reduced_pd(self, qpd):
        """Make a copy corresponding to a new plane-wave description."""
        new_chi0_optical_extension = Chi0OpticalExtensionData(self.wd, qpd)

        # Copy the head (present in any plane-wave representation)
        new_chi0_optical_extension.head_Wvv[:] = self.head_Wvv

        # Map the wings to the reduced plane-wave description
        G2_G1 = PWMapping(qpd, self.qpd).G2_G1
        new_chi0_optical_extension.wings_WxvG[:] \
            = self.wings_WxvG[..., G2_G1]

        return new_chi0_optical_extension


class Chi0Data(Chi0RelatedData):
    """Container object for the chi0 data objects for a single q-point,
    while holding also the corresponding basis descriptors and block
    distributor."""

    def __init__(self,
                 chi0_body: Chi0BodyData,
                 chi0_opt_ext: Chi0OpticalExtensionData | None = None):
        super().__init__(chi0_body.wd, chi0_body.qpd)
        self.body = chi0_body

        self.optical_limit = self.qpd.optical_limit
        if self.optical_limit:
            assert isinstance(chi0_opt_ext, Chi0OpticalExtensionData)
            assert chi0_opt_ext.wd is self.wd
            assert chi0_opt_ext.qpd is self.qpd
        else:
            assert chi0_opt_ext is None
        self.optical_extension = chi0_opt_ext

    @staticmethod
    def from_chi0_body_data(chi0_body):
        """Construct the container from a chi0 body data instance."""
        qpd = chi0_body.qpd
        if qpd.optical_limit:
            wd = chi0_body.wd
            chi0_optical_extension = Chi0OpticalExtensionData(wd, qpd)
        else:
            chi0_optical_extension = None

        return Chi0Data(chi0_body, chi0_optical_extension)

    def copy_with_reduced_pd(self, qpd):
        """Make a copy of the data object, reducing the plane wave basis."""
        new_body = self.body.copy_with_reduced_pd(qpd)
        if self.optical_limit:
            new_optical_extension = \
                self.optical_extension.copy_with_reduced_pd(qpd)
        else:
            new_optical_extension = None
        return Chi0Data(new_body, new_optical_extension)

    @property
    def chi0_WgG(self):
        return self.body.data_WgG

    @property
    def chi0_Wvv(self):
        if self.optical_limit:
            return self.optical_extension.head_Wvv

    @property
    def chi0_WxvG(self):
        if self.optical_limit:
            return self.optical_extension.wings_WxvG
```