```
from pathlib import Path
import numpy as np

from ase.units import Hartree

from gpaw.kpt_descriptor import KPointDescriptor
from gpaw.pw.descriptor import PWDescriptor

from gpaw.response.frequencies import ComplexFrequencyDescriptor
from gpaw.response.pair_integrator import DynamicPairFunction
from gpaw.response.pw_parallelization import (Blocks1D,
                                              PlaneWaveBlockDistributor)


class SingleQPWDescriptor(PWDescriptor):

    @staticmethod
    def from_q(q_c, ecut, gd, gammacentered=False):
        """Construct a plane wave descriptor for q_c with a given cutoff."""
        qd = KPointDescriptor([q_c])
        return SingleQPWDescriptor(ecut, gd, complex, qd,
                                   gammacentered=gammacentered)

    @property
    def q_c(self):
        return self.kd.bzk_kc[0]

    @property
    def optical_limit(self):
        return np.allclose(self.q_c, 0.0)

    def copy(self):
        return self.copy_with()

    def copy_with(self, ecut=None, gd=None, gammacentered=None):
        if ecut is None:
            ecut = self.ecut
        if gd is None:
            gd = self.gd
        if gammacentered is None:
            gammacentered = self.gammacentered

        return SingleQPWDescriptor.from_q(
            self.q_c, ecut, gd, gammacentered=gammacentered)


class LatticePeriodicPairFunction(DynamicPairFunction):
    r"""Data object for lattice periodic pair functions.

    Any spatial dependent pair function is considered to be lattice periodic,
    if it is invariant under translations of Bravais lattice vectors R:

    pf(r, r', z) = pf(r + R, r' + R, z).

    The Bloch lattice Fourier transform of a lattice periodic pair function,

                         __
                         \
    pf(r, r', q, z)  =   /_  e^(-iq.[r-r'-R']) pf(r, r' + R', z)

                         R'

    is then periodic in both r and r' independently and can be expressed in an
    arbitrary lattice periodic basis.

    In the GPAW response code, lattice periodic pair functions are expanded in
    plane waves,

                   1   //
    pf_GG'(q, z) = ‾  || drdr' e^(-iG.r) pf(r, r', q, z) e^(iG'.r')
```

```
                V0 //
                     V0

    which are encoded in the SingleQPWDescriptor along with the wave vector q.
    """

    def __init__(self,
                 qpd: SingleQPWDescriptor,
                 zd: ComplexFrequencyDescriptor,
                 blockdist: PlaneWaveBlockDistributor,
                 distribution='ZgG'):
        """Contruct the LatticePeriodicPairFunction.

        Parameters
        ----------
        distribution : str
            Memory distribution of the pair function array.
            Choices: 'ZgG', 'GZg' and 'zGG'.
        """
        self.qpd = qpd
        self.blockdist = blockdist
        self.distribution = distribution

        self.blocks1d = None
        self.shape = None
        super().__init__(qpd.q_c, zd)

    def zeros(self):
        if self.shape is None:
            self._initialize_block_distribution()
        return np.zeros(self.shape, complex)

    def _initialize_block_distribution(self):
        """Initialize 1D block distribution and corresponding array shape."""
        nz = len(self.zd)
        nG = self.qpd.ngmax
        blockdist = self.blockdist
        distribution = self.distribution

        if distribution == 'ZgG':
            blocks1d = Blocks1D(blockdist.blockcomm, nG)
            shape = (nz, blocks1d.nlocal, nG)
        elif distribution == 'GZg':
            blocks1d = Blocks1D(blockdist.blockcomm, nG)
            shape = (nG, nz, blocks1d.nlocal)
        elif distribution == 'zGG':
            blocks1d = Blocks1D(blockdist.blockcomm, nz)
            shape = (blocks1d.nlocal, nG, nG)
        else:
            raise NotImplementedError(f'Distribution: {distribution}')

        self.blocks1d = blocks1d
        self.shape = shape

    def array_with_view(self, view):
        """Access a given view into the pair function array."""
        if view == 'ZgG' and self.distribution in ['ZgG', 'GZg']:
            if self.distribution == 'GZg':
                pf_GZg = self.array
                pf_ZgG = pf_GZg.transpose((1, 2, 0))
            else:
                pf_ZgG = self.array

            pf_x = pf_ZgG
        else:
            raise ValueError(f'{view} is not a valid view, when array is of '
                             f'distribution {self.distribution}')

        return pf_x
```

```
    def copy_with_distribution(self, distribution='ZgG'):
        """Copy the pair function to a specified memory distribution."""
        new_pf = self._new(*self.my_args(), distribution=distribution)
        new_pf.array[:] = self.array_with_view(distribution)

        return new_pf

    @classmethod
    def _new(cls, *args, **kwargs):
        return cls(*args, **kwargs)

    def my_args(self, qpd=None, zd=None, blockdist=None):
        """Return the positional construction arguments of the
        LatticePeriodicPairFunction."""
        if qpd is None:
            qpd = self.qpd
        if zd is None:
            zd = self.zd
        if blockdist is None:
            blockdist = self.blockdist

        return qpd, zd, blockdist

    def copy_with_reduced_pd(self, qpd):
        """Copy the pair function, but within a reduced plane-wave basis."""
        new_pf = self._new(*self.my_args(qpd=qpd),
                           distribution=self.distribution)
        if self.distribution == 'zGG':
            new_pf.array[:] = map_zGG_array_to_reduced_pd(self.qpd, qpd,
                                                          self.array)
        elif self.distribution == 'ZgG':
            new_pf.array[:] = map_ZgG_array_to_reduced_pd(self.qpd, qpd,
                                                          self.blockdist,
                                                          self.array)
        else:
            raise NotImplementedError('Chi.copy_with_reduced_pd has not been '
                                      'implemented for distribution '
                                      f'{self.distribution}')
        return new_pf

    def copy_with_global_frequency_distribution(self):
        """Copy the pair function, but with distribution zGG over world."""
        # Make a copy, which is globally block distributed
        blockdist = self.blockdist.new_distributor(nblocks='max')
        new_pf = self._new(*self.my_args(blockdist=blockdist),
                           distribution='zGG')

        # Redistribute the data, distributing the frequencies over world
        assert self.distribution == 'ZgG'
        new_pf.array[:] = self.blockdist.distribute_frequencies(self.array,
                                                                len(self.zd))

        return new_pf


def map_ZgG_array_to_reduced_pd(qpdi, qpd, blockdist, in_ZgG):
    """Map the array in_ZgG from the qpdi to the qpd plane-wave basis."""
    # Distribute over frequencies
    nw = in_ZgG.shape[0]
    tmp_zGG = blockdist.distribute_as(in_ZgG, nw, 'zGG')

    # Reduce the plane-wave basis
    tmp_zGG = map_zGG_array_to_reduced_pd(qpdi, qpd, tmp_zGG)

    # Distribute over plane waves
    out_ZgG = blockdist.distribute_as(tmp_zGG, nw, 'ZgG')

    return out_ZgG
```

```
def map_zGG_array_to_reduced_pd(qpdi, qpd, in_zGG):
    """Map the array in_zGG from the qpdi to the qpd plane-wave basis."""
    from gpaw.pw.descriptor import PWMapping

    # Initialize the basis mapping
    pwmapping = PWMapping(qpdi, qpd)
    G2_GG = tuple(np.meshgrid(pwmapping.G2_G1, pwmapping.G2_G1,
                              indexing='ij'))
    G1_GG = tuple(np.meshgrid(pwmapping.G1, pwmapping.G1,
                              indexing='ij'))

    # Allocate array in the new basis
    nG = qpd.ngmax
    out_zGG_shape = (in_zGG.shape[0], nG, nG)
    out_zGG = np.zeros(out_zGG_shape, complex)

    # Extract values
    for z, in_GG in enumerate(in_zGG):
        out_zGG[z][G2_GG] = in_GG[G1_GG]

    return out_zGG


class Chi(LatticePeriodicPairFunction):
    r"""Data object for the four-component susceptibility tensor χ_GG'^μν(q,z).
    """

    def __init__(self, spincomponent, qpd, zd,
                 blockdist, distribution='ZgG'):
        r"""Construct a susceptibility of a given spin-component (μν)."""
        self.spincomponent = spincomponent
        super().__init__(qpd, zd, blockdist, distribution=distribution)

    def new(self, **kwargs):
        return self._new(*self.my_args_and_kwargs(**kwargs))

    def my_args(self, spincomponent=None, qpd=None, zd=None, blockdist=None):
        """Return positional construction arguments of the Chi object."""
        if spincomponent is None:
            spincomponent = self.spincomponent
        qpd, zd, blockdist = super().my_args(qpd=qpd, zd=zd,
                                             blockdist=blockdist)

        return spincomponent, qpd, zd, blockdist

    def my_args_and_kwargs(self, distribution=None, **args):
        """Return all the construction arguments of Chi, in order."""
        args = self.my_args(**args)
        if distribution is None:
            distribution = self.distribution
        return args + (distribution,)

    def copy_with_reduced_ecut(self, ecut):
        """Copy the susceptibility, but with a reduced ecut."""
        ecut = ecut / Hartree  # eV -> Hartree
        assert ecut <= self.qpd.ecut
        qpd = self.qpd.copy_with(ecut=ecut)
        return self.copy_with_reduced_pd(qpd)

    def copy_reactive_part(self):
        r"""Return a copy of the reactive part of the susceptibility.

        The reactive part of the susceptibility is defined as (see
        [PRB 103, 245110 (2021)]):

                              1
        χ_GG'^(μν')(q,z) = ―― [χ_GG'^μν(q,z) + χ_(-G'-G)^νμ(-q,-z*)].
```

```
                    2

    However if the density operators n^μ(r) and n^ν(r) are each others
    Hermitian conjugates, the reactive part simply becomes the Hermitian
    part in terms of the plane-wave basis:

                                1
    χ_GG'^(μν')(q,z) = ─ [χ_GG'^μν(q,z) + χ_G'G^(μν*)(q,z)],
                                2

    which is trivial to evaluate.
    """
    assert self.distribution == 'zGG' or \
        (self.distribution == 'ZgG' and self.blockdist.blockcomm.size == 1)
    assert self.spincomponent in ['00', 'uu', 'dd', '+-', '-+'], \
        'Spin-density operators has to be each others hermitian conjugates'
    chiksr = self.new(distribution='zGG')
    chiks_zGG = self.array
    chiksr.array += chiks_zGG
    chiksr.array += np.conj(np.transpose(chiks_zGG, (0, 2, 1)))
    chiksr.array /= 2.

    return chiksr

def copy_dissipative_part(self):
    r"""Return a copy of the dissipative part of the susceptibility.

    The dissipative part of the susceptibility is defined as (see
    [PRB 103, 245110 (2021)]):

                                1
    χ_GG'^(μν")(q,z) = ─── [χ_GG'^μν(q,z) - χ_(-G'-G)^νμ(-q,-z*)].
                               2i

    Similar to the reactive part, this expression reduces to the
    anti-Hermitian part of the susceptibility in terms of the plane-wave
    basis, whenever the density operators n^μ(r) and n^ν(r) are each others
    Hermitian conjugates:

                                1
    χ_GG'^(μν")(q,z) = ─── [χ_GG'^μν(q,z) - χ_G'G^(μν*)(q,z)].
                               2i
    """
    assert self.distribution == 'zGG' or \
        (self.distribution == 'ZgG' and self.blockdist.blockcomm.size == 1)
    assert self.spincomponent in ['00', 'uu', 'dd', '+-', '-+'], \
        'Spin-density operators has to be each others hermitian conjugates'
    chiksd = self.new(distribution='zGG')
    chiks_zGG = self.array
    chiksd.array += chiks_zGG
    chiksd.array -= np.conj(np.transpose(chiks_zGG, (0, 2, 1)))
    chiksd.array /= 2.j

    return chiksd

def symmetrize_reciprocity(self):
    r"""Symmetrize the reciprocity of the susceptibility (for q=0).

    In collinear systems without spin-orbit coupling, the plane-wave
    susceptibility is reciprocal in the sense that

    χ_GG'^(μν)(q, ω) = χ_(-G'-G)^(μν)(-q, ω)

    for all μν ∈ {00, uu, dd, +-, -+}, see [PRB 106, 085131 (2022)].
    For q=0, we may symmetrize the susceptibility in this sense for free.
    """
    assert np.allclose(self.q_c, 0.)
    assert self.distribution == 'zGG' or \
        (self.distribution == 'ZgG' and self.blockdist.blockcomm.size == 1)
```

```
        assert self.spincomponent in ['00', 'uu', 'dd', '+-', '-+'], \
            'Spin-density operators has to be each others hermitian conjugates'
        invmap_GG = get_inverted_pw_mapping(self.qpd, self.qpd)
        for chi_GG in self.array:
            # Symmetrize [χ_(GG')(q, ω) + χ_(-G'-G)(-q, ω)] / 2
            chi_GG[:] = (chi_GG + chi_GG[invmap_GG].T) / 2.

    def write_macroscopic_component(self, filename):
        """Write the spatially averaged (macroscopic) component of the
        susceptibility to a file along with the frequency grid."""
        chi_Z = self.get_macroscopic_component()
        if self.blocks1d.blockcomm.rank == 0:
            write_pair_function(filename, self.zd, chi_Z)

    def get_macroscopic_component(self):
        """Get the macroscopic (G=0) component, all-gathered."""
        assert self.distribution == 'zGG'
        chi_zGG = self.array
        chi_z = chi_zGG[:, 0, 0]  # Macroscopic component
        chi_Z = self.blocks1d.all_gather(chi_z)
        return chi_Z

    def write_array(self, filename):
        """Write the full susceptibility array to a file along with the
        frequency grid and plane-wave components."""
        assert self.distribution == 'zGG'
        chi_ZGG = self.blocks1d.gather(self.array)
        if self.blocks1d.blockcomm.rank == 0:
            write_susceptibility_array(filename, self.zd, self.qpd, chi_ZGG)

    def write_diagonal(self, filename):
        """Write the diagonal of the many-body susceptibility within a reduced
        plane-wave basis to a file along with the frequency grid."""
        assert self.distribution == 'zGG'
        chi_zGG = self.array
        chi_zG = np.diagonal(chi_zGG, axis1=1, axis2=2)
        chi_ZG = self.blocks1d.gather(chi_zG)
        if self.blocks1d.blockcomm.rank == 0:
            write_susceptibility_array(filename, self.zd, self.qpd, chi_ZG)


def get_inverted_pw_mapping(qpd1, qpd2):
    """Get the planewave coefficients mapping GG' of qpd1 into -G-G' of qpd2"""
    G1_Gc = get_pw_coordinates(qpd1)
    G2_Gc = get_pw_coordinates(qpd2)

    mG2_G1 = []
    for G1_c in G1_Gc:
        found_match = False
        for G2, G2_c in enumerate(G2_Gc):
            if np.all(G2_c == -G1_c):
                mG2_G1.append(G2)
                found_match = True
                break
        if not found_match:
            raise ValueError('Could not match qpd1 and qpd2')

    # Set up mapping from GG' to -G-G'
    invmap_GG = tuple(np.meshgrid(mG2_G1, mG2_G1, indexing='ij'))

    return invmap_GG


def get_pw_coordinates(qpd):
    """Get the reciprocal lattice vector coordinates corresponding to a
    givne plane wave basis.

    Please remark, that the response code currently works with one q-vector
    at a time, at thus only a single plane wave representation at a time.
```

```
    Returns
    -------
    G_Gc : nd.array (dtype=int)
        Coordinates on the reciprocal lattice
    """
    # List of all plane waves
    G_Gv = np.array([qpd.G_Qv[Q] for Q in qpd.Q_qG[0]])

    # Use cell to get coordinates
    B_cv = 2.0 * np.pi * qpd.gd.icell_cv
    return np.round(np.dot(G_Gv, np.linalg.inv(B_cv))).astype(int)


def write_pair_function(filename, zd, pf_z):
    """Write a pair function pf(q,z) for a specific q."""
    # For now, we assume that the complex frequencies lie on a horizontal
    # contour and that the pair function is complex. This could be easily
    # generalized at a later stage.
    assert zd.horizontal_contour
    omega_w = zd.omega_w * Hartree  # Ha -> eV
    pf_w = pf_z  # Atomic units
    assert pf_w.dtype == complex

    # Write results file
    with open(filename, 'w') as fd:
        print('# {:>11}, {:>11}, {:>11}'.format(
            'omega [eV]', 'pf_w.real', 'pf_w.imag'), file=fd)
        for omega, pf in zip(omega_w, pf_w):
            print('  {:11.6f}, {:11.6f}, {:11.6f}'.format(
                omega, pf.real, pf.imag), file=fd)


def read_pair_function(filename):
    """Read a stored pair function file."""
    d = np.loadtxt(filename, delimiter=',')

    if d.shape[1] == 3:
        # Complex pair function on a horizontal contour
        omega_w = np.array(d[:, 0], float)
        pf_w = np.array(d[:, 1], complex)
        pf_w.imag = d[:, 2]
    else:
        raise ValueError(f'Unexpected array dimension {d.shape}')

    return omega_w, pf_w


def write_susceptibility_array(filename, zd, qpd, chi_zx):
    """Write dynamic susceptibility array to a .npz file."""
    assert Path(filename).suffix == '.npz', filename
    # For now, we assume that the complex frequencies lie on a horizontal
    # contour
    assert zd.horizontal_contour
    omega_w = zd.omega_w * Hartree  # Ha -> eV
    G_Gc = get_pw_coordinates(qpd)
    chi_wx = chi_zx
    np.savez(filename, omega_w=omega_w, G_Gc=G_Gc, chi_wx=chi_wx)


def read_susceptibility_array(filename):
    """Read a stored dynamic susceptibility array file."""
    assert Path(filename).suffix == '.npz', filename
    npzfile = np.load(filename)
    return npzfile['omega_w'], npzfile['G_Gc'], npzfile['chi_wx']
```