```
from __future__ import annotations
from abc import ABC, abstractmethod

import numpy as np

from typing import TYPE_CHECKING

from ase.units import Ha
from gpaw.bztools import convex_hull_volume
from gpaw.response import timer
from gpaw.response.pair import KPointPairFactory
from gpaw.response.frequencies import NonLinearFrequencyDescriptor
from gpaw.response.pair_functions import SingleQPWDescriptor
from gpaw.response.pw_parallelization import block_partition
from gpaw.response.integrators import (
    Integrand, PointIntegrator, TetrahedronIntegrator, Domain)
from gpaw.response.symmetry import QSymmetryInput, QSymmetryAnalyzer
from gpaw.response.kpoints import KPointDomain, KPointDomainGenerator

if TYPE_CHECKING:
    from gpaw.response.pair import ActualPairDensityCalculator
    from gpaw.response.context import ResponseContext
    from gpaw.response.groundstate import ResponseGroundStateAdapter


class Chi0Integrand(Integrand):
    def __init__(self, chi0calc: Chi0ComponentPWCalculator,
                 optical: bool,
                 qpd: SingleQPWDescriptor,
                 generator: KPointDomainGenerator,
                 m1: int,
                 m2: int):

        self._chi0calc = chi0calc

        # In a normal response calculation, we include transitions from all
        # completely and partially unoccupied bands to range(m1, m2)

        self.gs: ResponseGroundStateAdapter = chi0calc.gs
        self.n1 = 0
        self.n2 = self.gs.nocc2
        assert m1 <= m2
        self.m1 = m1
        self.m2 = m2

        self.context: ResponseContext = chi0calc.context
        self.kptpair_factory: KPointPairFactory = chi0calc.kptpair_factory

        self.qpd = qpd
        self.generator = generator
        self.integrationmode = chi0calc.integrationmode
        self.optical = optical
        self.blockcomm = chi0calc.blockcomm

    @timer('Get matrix element')
    def matrix_element(self, point):
        """Return pair density matrix element for integration.

        A pair density is defined as::

         <snk| e^(-i (q + G) r) |s'mk+q>,

        where s and s' are spins, n and m are band indices, k is
        the kpoint and q is the momentum transfer. For dielectric
        response s'=s, for the transverse magnetic response
        s' is flipped with respect to s.

        Parameters
        ----------
```

```
        k_v : ndarray
            Kpoint coordinate in cartesian coordinates.
        s : int
            Spin index.

        If self.optical, then return optical pair densities, that is, the
        head and wings matrix elements indexed by:
        # P = (x, y, v, G1, G2, ...).

        Return
        ------
        n_nmG : ndarray
            Pair densities.
        """

        if self.optical:
            # pair_calc: ActualPairDensityCalculator from gpaw.response.pair
            target_method = self._chi0calc.pair_calc.get_optical_pair_density
            out_ngmax = self.qpd.ngmax + 2
        else:
            target_method = self._chi0calc.pair_calc.get_pair_density
            out_ngmax = self.qpd.ngmax

        return self._get_any_matrix_element(
            point, target_method=target_method,
        ).reshape(-1, out_ngmax)

    def _get_any_matrix_element(self, point, target_method):
        qpd = self.qpd

        k_v = point.kpt_c  # XXX c/v discrepancy

        k_c = np.dot(qpd.gd.cell_cv, k_v) / (2 * np.pi)
        K = self.gs.kpoints.kptfinder.find(k_c)
        # assert point.K == K, (point.K, K)

        weight = np.sqrt(self.generator.get_kpoint_weight(k_c) /
                         self.generator.how_many_symmetries())

        # Here we're again setting pawcorr willy-nilly
        if self._chi0calc.pawcorr is None:
            pairden_paw_corr = self.gs.pair_density_paw_corrections
            self._chi0calc.pawcorr = pairden_paw_corr(qpd)

        kptpair = self.kptpair_factory.get_kpoint_pair(
            qpd, point.spin, K, self.n1, self.n2,
            self.m1, self.m2, blockcomm=self.blockcomm)

        m_m = np.arange(self.m1, self.m2)
        n_n = np.arange(self.n1, self.n2)
        n_nmG = target_method(qpd, kptpair, n_n, m_m,
                              pawcorr=self._chi0calc.pawcorr,
                              block=True)

        if self.integrationmode is None:
            n_nmG *= weight

        df_nm = kptpair.get_occupation_differences()
        df_nm[df_nm <= 1e-20] = 0.0
        n_nmG *= df_nm[..., np.newaxis]**0.5

        return n_nmG

    @timer('Get eigenvalues')
    def eigenvalues(self, point):
        """A function that can return the eigenvalues.

        A simple function describing the integrand of
        the response function which gives an output that
```

```
        is compatible with the gpaw k-point integration
        routines."""

        qpd = self.qpd
        gs = self.gs
        kd = gs.kd

        k_v = point.kpt_c  # XXX c/v discrepancy

        k_c = np.dot(qpd.gd.cell_cv, k_v) / (2 * np.pi)
        kptfinder = self.gs.kpoints.kptfinder
        K1 = kptfinder.find(k_c)
        K2 = kptfinder.find(k_c + qpd.q_c)

        ik1 = kd.bz2ibz_k[K1]
        ik2 = kd.bz2ibz_k[K2]
        kpt1 = gs.kpt_qs[ik1][point.spin]
        assert kd.comm.size == 1
        kpt2 = gs.kpt_qs[ik2][point.spin]
        deps_nm = np.subtract(kpt1.eps_n[self.n1:self.n2][:, np.newaxis],
                              kpt2.eps_n[self.m1:self.m2])
        return deps_nm.reshape(-1)


class Chi0ComponentCalculator:
    """Base class for the Chi0XXXCalculator suite."""

    def __init__(self, gs, context, *, nblocks,
                 qsymmetry: QSymmetryInput = True,
                 integrationmode=None):
        """Set up attributes common to all chi0 related calculators.

        Parameters
        ----------
        nblocks : int
            Divide response function memory allocation in nblocks.
        qsymmetry: bool, dict, or QSymmetryAnalyzer
            QSymmetryAnalyzer, or bool to enable all/no symmetries,
            or dict with which to create QSymmetryAnalyzer.
            Disabling symmetries may be useful for debugging.
        integrationmode : str or None
            Integrator for the k-point integration.
            If == 'tetrahedron integration' then the kpoint integral is
            performed using the linear tetrahedron method. If None, point
            integration is used.
        """
        self.gs = gs
        self.context = context
        self.kptpair_factory = KPointPairFactory(gs, context)

        self.nblocks = nblocks
        self.blockcomm, self.kncomm = block_partition(
            self.context.comm, self.nblocks)

        self.qsymmetry = QSymmetryAnalyzer.from_input(qsymmetry)

        # Set up integrator
        self.integrationmode = integrationmode
        self.integrator = self.construct_integrator()

    @property
    def pbc(self):
        return self.gs.pbc

    def construct_integrator(self):  # -> Integrator or child of Integrator
        """Construct k-point integrator"""
        cls = self.get_integrator_cls()
        return cls(
            cell_cv=self.gs.gd.cell_cv,
```

```
            context=self.context,
            blockcomm=self.blockcomm,
            kncomm=self.kncomm)

    def get_integrator_cls(self):  # -> Integrator or child of Integrator
        """Get the appointed k-point integrator class."""
        if self.integrationmode is None:
            self.context.print('Using integrator: PointIntegrator')
            cls = PointIntegrator
        elif self.integrationmode == 'tetrahedron integration':
            self.context.print('Using integrator: TetrahedronIntegrator')
            cls = TetrahedronIntegrator
            if not self.qsymmetry.disabled:
                self.check_high_symmetry_ibz_kpts()
        else:
            raise ValueError(f'Integration mode "{self.integrationmode}"'
                             ' not implemented.')
        return cls

    def check_high_symmetry_ibz_kpts(self):
        """Check that the ground state includes all corners of the IBZ."""
        ibz_vertices_kc = self.gs.get_ibz_vertices()
        # Here we mimic the k-point grid compatibility check of
        # gpaw.bztools.find_high_symmetry_monkhorst_pack()
        bzk_kc = self.gs.kd.bzk_kc
        for ibz_vertex_c in ibz_vertices_kc:
            # Relative coordinate difference to the k-point grid
            diff_kc = np.abs(bzk_kc - ibz_vertex_c)[:, self.gs.pbc].round(6)
            # The ibz vertex should exits in the BZ grid up to a reciprocal
            # lattice vector, meaning that the relative coordinate difference
            # is allowed to be an integer. Thus, at least one relative k-point
            # difference should vanish, modulo 1
            mod_diff_kc = np.mod(diff_kc, 1)
            nodiff_k = np.all(mod_diff_kc < 1e-5, axis=1)
            if not np.any(nodiff_k):
                raise ValueError(
                    'The ground state k-point grid does not include all '
                    'vertices of the IBZ. '
                    'Please use find_high_symmetry_monkhorst_pack() from '
                    'gpaw.bztools to generate your k-point grid.')

    def get_integration_domain(self, q_c, spins):
        """Get integrator domain and prefactor for the integral."""
        for spin in spins:
            assert spin in range(self.gs.nspins)
        # The integration domain is determined by the following function
        # that reduces the integration domain to the irreducible zone
        # of the little group of q.
        symmetries, generator, kpoints = self.get_kpoints(
            q_c, integrationmode=self.integrationmode)

        domain = Domain(kpoints.k_kv, spins)

        if self.integrationmode == 'tetrahedron integration':
            # If there are non-periodic directions it is possible that the
            # integration domain is not compatible with the symmetry operations
            # which essentially means that too large domains will be
            # integrated. We normalize by vol(BZ) / vol(domain) to make
            # sure that to fix this.
            domainvol = convex_hull_volume(
                kpoints.k_kv) * generator.how_many_symmetries()
            bzvol = (2 * np.pi)**3 / self.gs.volume
            factor = bzvol / domainvol
        else:
            factor = 1

        prefactor = (2 * factor * generator.how_many_symmetries() /
                     (self.gs.nspins * (2 * np.pi)**3))  # Remember prefactor
```

```
        if self.integrationmode is None:
            nbzkpts = self.gs.kd.nbzkpts
            prefactor *= len(kpoints) / nbzkpts

        return symmetries, generator, domain, prefactor

    @timer('Get kpoints')
    def get_kpoints(self, q_c, integrationmode):
        """Get the integration domain."""
        symmetries, generator = self.qsymmetry.analyze(
            np.asarray(q_c), self.gs.kpoints, self.context)

        if integrationmode is None:
            k_kc = generator.get_kpt_domain()
        elif integrationmode == 'tetrahedron integration':
            k_kc = generator.get_tetrahedron_kpt_domain(
                pbc_c=self.pbc, cell_cv=self.gs.gd.cell_cv)
        kpoints = KPointDomain(k_kc, self.gs.gd.icell_cv)

        # In the future, we probably want to put enough functionality on the
        # KPointDomain such that we don't need to also return the
        # KPointDomainGenerator XXX
        return symmetries, generator, kpoints

    def get_gs_info_string(self, tab=''):
        gs = self.gs
        gd = gs.gd

        ns = gs.nspins
        nk = gs.kd.nbzkpts
        nik = gs.kd.nibzkpts

        nocc = self.gs.nocc1
        npocc = self.gs.nocc2
        ngridpoints = gd.N_c[0] * gd.N_c[1] * gd.N_c[2]
        nstat = ns * npocc
        occsize = nstat * ngridpoints * 16. / 1024**2

        gs_list = [f'{tab}Ground state adapter containing:',
                   f'Number of spins: {ns}', f'Number of kpoints: {nk}',
                   f'Number of irreducible kpoints: {nik}',
                   f'Number of completely occupied states: {nocc}',
                   f'Number of partially occupied states: {npocc}',
                   f'Occupied states memory: {occsize} M / cpu']

        return f'\n{tab}'.join(gs_list)


class Chi0ComponentPWCalculator(Chi0ComponentCalculator, ABC):
    """Base class for Chi0XXXCalculators, which utilize a plane-wave basis."""

    def __init__(self, gs, context,
                 *,
                 wd,
                 hilbert=True,
                 nbands=None,
                 timeordered=False,
                 ecut=50.0,
                 eta=0.2,
                 **kwargs):
        """Set up attributes to calculate the chi0 body and optical extensions.

        Parameters
        ----------
        wd : FrequencyDescriptor
            Frequencies for which the chi0 component is evaluated.
        hilbert : bool
            Hilbert transform flag. If True, the dissipative part of the chi0
            component is evaluated, and the reactive part is calculated via a
```

```
            hilbert transform. Only works for frequencies on the real axis and
            requires a nonlinear frequency grid.
        nbands : int
            Number of bands to include.
        timeordered : bool
            Flag for calculating the time ordered chi0 component. Used for
            G0W0, which performs its own hilbert transform.
        ecut : float
            Plane-wave energy cutoff in eV.
        eta : float
            Artificial broadening of the chi0 component in eV.
        """
        super().__init__(gs, context, **kwargs)

        self.ecut = ecut / Ha
        self.nbands = nbands or self.gs.bd.nbands

        self.wd = wd
        self.context.print(self.wd, flush=False)

        self.eta = eta / Ha
        self.hilbert = hilbert
        self.task = self.construct_integral_task()

        self.timeordered = bool(timeordered)
        if self.timeordered:
            assert self.hilbert  # Timeordered is only needed for G0W0

        self.pawcorr = None

        self.context.print('Nonperiodic BCs: ', (~self.pbc))
        if sum(self.pbc) == 1:
            raise ValueError('1-D not supported atm.')

    @property
    def pair_calc(self) -> ActualPairDensityCalculator:
        return self.kptpair_factory.pair_calculator(self.blockcomm)

    def construct_integral_task(self):
        if self.eta == 0:
            assert not self.hilbert
            # eta == 0 is used as a synonym for calculating the hermitian part
            # of the response function at a range of imaginary frequencies
            assert not self.wd.omega_w.real.any()
            return self.construct_hermitian_task()

        if self.hilbert:
            # The hilbert flag is used to calculate the reponse function via a
            # hilbert transform of its dissipative (spectral) part.
            assert isinstance(self.wd, NonLinearFrequencyDescriptor)
            return self.construct_hilbert_task()

        # Otherwise, we perform a literal evaluation of the response function
        # at the given frequencies with broadening eta
        return self.construct_literal_task()

    @abstractmethod
    def construct_hermitian_task(self):
        """Integral task for the hermitian part of chi0."""

    def construct_hilbert_task(self):
        if isinstance(self.integrator, PointIntegrator):
            return self.construct_point_hilbert_task()
        else:
            assert isinstance(self.integrator, TetrahedronIntegrator)
            return self.construct_tetra_hilbert_task()

    @abstractmethod
    def construct_point_hilbert_task(self):
```

```
        """Integral task for point integrating the spectral part of chi0."""

    @abstractmethod
    def construct_tetra_hilbert_task(self):
        """Integral task for tetrahedron integration of the spectral part."""

    @abstractmethod
    def construct_literal_task(self):
        """Integral task for a literal evaluation of chi0."""

    def get_pw_descriptor(self, q_c):
        return SingleQPWDescriptor.from_q(q_c, self.ecut, self.gs.gd)

    def get_band_transitions(self):
        return self.gs.nocc1, self.nbands  # m1, m2

    def get_response_info_string(self, qpd, tab=''):
        nw = len(self.wd)
        ecut = self.ecut * Ha
        nbands = self.nbands
        ngmax = qpd.ngmax
        eta = self.eta * Ha

        res_list = [f'{tab}Number of frequency points: {nw}',
                    f'Planewave cutoff: {ecut}',
                    f'Number of bands: {nbands}',
                    f'Number of planewaves: {ngmax}',
                    f'Broadening (eta): {eta}']

        return f'\n{tab}'.join(res_list)
```