```python
from __future__ import annotations

from time import ctime
from typing import TYPE_CHECKING

import numpy as np
from ase.units import Ha

import gpaw
from gpaw.response import (ResponseGroundStateAdapter, ResponseContext,
                           ResponseGroundStateAdaptable, ResponseContextInput)
from gpaw.response.symmetrize import (BodySymmetryOperators,
                                      WingSymmetryOperators)
from gpaw.response.chi0_data import (Chi0Data, Chi0BodyData,
                                     Chi0OpticalExtensionData)
from gpaw.response.frequencies import FrequencyDescriptor
from gpaw.response.pair_functions import SingleQPWDescriptor
from gpaw.response.hilbert import HilbertTransform
from gpaw.response import timer
from gpaw.response.pw_parallelization import PlaneWaveBlockDistributor
from gpaw.utilities.memory import maxrss
from gpaw.response.chi0_base import Chi0ComponentPWCalculator, Chi0Integrand
from gpaw.response.integrators import (
    HermitianOpticalLimit, HilbertOpticalLimit, OpticalLimit,
    HilbertOpticalLimitTetrahedron,
    Hermitian, Hilbert, HilbertTetrahedron, GenericUpdate)

if TYPE_CHECKING:
    from typing import Any
    from gpaw.typing import ArrayLike1D
    from gpaw.response.pair import ActualPairDensityCalculator


class Chi0Calculator:
    def __init__(self,
                 gs: ResponseGroundStateAdaptable,
                 context: ResponseContextInput = '-',
                 nblocks=1,
                 eshift=None,
                 intraband=True,
                 rate=0.0,
                 **kwargs):
        self.gs = ResponseGroundStateAdapter.from_input(gs)
        self.context = ResponseContext.from_input(context)

        self.chi0_body_calc = Chi0BodyCalculator(
            self.gs, self.context,
            nblocks=nblocks, eshift=eshift, **kwargs)
        self.chi0_opt_ext_calc = Chi0OpticalExtensionCalculator(
            self.gs, self.context,
            intraband=intraband, rate=rate, **kwargs)

    @property
    def wd(self) -> FrequencyDescriptor:
        wd = self.chi0_body_calc.wd
        assert wd is self.chi0_opt_ext_calc.wd
        return wd

    @property
    def pair_calc(self) -> ActualPairDensityCalculator:
        # In a future refactor, we should find better ways to access the pair
        # density calculator (and the pair density paw corrections) XXX

        # pair_calc: ActualPairDensityCalculator from gpaw.response.pair
        return self.chi0_body_calc.pair_calc

    def create_chi0(self, q_c: list | np.ndarray) -> Chi0Data:

        # chi0_body: Chi0BodyData from gpaw.response.chi0_data
```

```python
        chi0_body = self.chi0_body_calc.create_chi0_body(q_c)

        # chi0: Chi0Data from gpaw.response.chi0_data
        chi0 = Chi0Data.from_chi0_body_data(chi0_body)
        return chi0

    def calculate(self, q_c: list | np.ndarray) -> Chi0Data:
        """Calculate chi0 (possibly with optical extensions).

        Parameters
        ----------
        q_c : list or ndarray
            Momentum vector.

        Returns
        -------
        chi0 : Chi0Data
            Data object containing the chi0 data arrays along with basis
            representation descriptors and blocks distribution
        """
        # Calculate body

        # chi0_body: Chi0BodyData from gpaw.response.chi0_data
        chi0_body = self.chi0_body_calc.calculate(q_c)
        # SingleQPWDescriptor from gpaw.response.pair_functions
        qpd = chi0_body.qpd

        # Calculate optical extension
        if qpd.optical_limit:
            if self.chi0_body_calc.eshift is not None:
                raise NotImplementedError("No wings eshift available")
            chi0_opt_ext = self.chi0_opt_ext_calc.calculate(qpd=qpd)
        else:
            chi0_opt_ext = None

        self.context.print('\nFinished calculating chi0\n')

        return Chi0Data(chi0_body, chi0_opt_ext)

    @timer('Calculate CHI_0')
    def update_chi0(self,
                    chi0: Chi0Data,
                    m1: int, m2: int, spins: list) -> Chi0Data:
        """In-place calculation of chi0 (with optical extension).

        Parameters
        ----------
        chi0 : Chi0Data
            Data and representation object
        m1 : int
            Lower band cutoff for band summation
        m2 : int
            Upper band cutoff for band summation
        spins : list
            List of spin indices to include in the calculation

        Returns
        -------
        chi0 : Chi0Data
        """
        self.chi0_body_calc.update_chi0_body(chi0.body, m1, m2, spins)
        if chi0.optical_limit:
            if self.chi0_body_calc.eshift is not None:
                raise NotImplementedError("No wings eshift available")
            assert chi0.optical_extension is not None
            # Update the head and wings
            self.chi0_opt_ext_calc.update_chi0_optical_extension(
                chi0.optical_extension, m1, m2, spins)
        return chi0
```

```python
class Chi0BodyCalculator(Chi0ComponentPWCalculator):
    def __init__(self, *args,
                 eshift: float | None = None,
                 **kwargs):
        """Construct the Chi0BodyCalculator.

        Parameters
        ----------
        eshift : float or None
            Energy shift of the conduction bands in eV.
        """
        self.eshift = eshift / Ha if eshift else eshift

        super().__init__(*args, **kwargs)

        if self.gs.metallic:
            assert self.eshift is None, \
                'A rigid energy shift cannot be applied to the conduction '\
                'bands if there is no band gap'

    def create_chi0_body(self, q_c: list | np.ndarray) -> Chi0BodyData:
        # qpd: SingleQPWDescriptor from gpaw.response.pair_functions
        qpd = self.get_pw_descriptor(q_c)
        return self._create_chi0_body(qpd)

    def _create_chi0_body(self, qpd: SingleQPWDescriptor) -> Chi0BodyData:
        return Chi0BodyData(self.wd, qpd, self.get_blockdist())

    def get_blockdist(self) -> PlaneWaveBlockDistributor:
        # integrator: Integrator from gpaw.response.integrators
        #     (or a child of this class)
        return PlaneWaveBlockDistributor(
            self.context.comm,  # _Communicator object from gpaw.mpi
            self.integrator.blockcomm,  # _Communicator object from gpaw.mpi
            self.integrator.kncomm)  # _Communicator object from gpaw.mpi

    def calculate(self, q_c: list | np.ndarray) -> Chi0BodyData:
        """Calculate the chi0 body.

        Parameters
        ----------
        q_c : list or ndarray
            Momentum vector.
        """
        # Construct the output data structure
        # qpd: SingleQPWDescriptor from gpaw.response.pair_functions
        qpd = self.get_pw_descriptor(q_c)
        self.print_info(qpd)
        # chi0_body: Chi0BodyData from gpaw.response.chi0_data
        chi0_body = self._create_chi0_body(qpd)

        # Integrate all transitions into partially filled and empty bands
        m1, m2 = self.get_band_transitions()
        self.update_chi0_body(chi0_body, m1, m2, spins=range(self.gs.nspins))

        return chi0_body

    def update_chi0_body(self,
                         chi0_body: Chi0BodyData,
                         m1: int, m2: int, spins: list | range):
        """In-place calculation of the body.

        Parameters
        ----------
        m1 : int
            Lower band cutoff for band summation
        m2 : int
```

```
            Upper band cutoff for band summation
        spins : list
            List of spin indices to include in the calculation
        """
        qpd = chi0_body.qpd

        # Reset PAW correction in case momentum has change
        pairden_paw_corr = self.gs.pair_density_paw_corrections
        self.pawcorr = pairden_paw_corr(chi0_body.qpd)

        self.context.print('Integrating chi0 body.')

        # symmetries: QSymmetries from gpaw.response.symmetry
        # generator: KPointDomainGenerator from gpaw.response.kpoints
        # domain: Domain from from gpaw.response.integrators
        symmetries, generator, domain, prefactor = self.get_integration_domain(
            qpd.q_c, spins)
        integrand = Chi0Integrand(self, qpd=qpd, generator=generator,
                                  optical=False, m1=m1, m2=m2)

        chi0_body.data_WgG[:] /= prefactor
        if self.hilbert:
            # Allocate a temporary array for the spectral function
            out_WgG = chi0_body.zeros()
        else:
            # Use the preallocated array for direct updates
            out_WgG = chi0_body.data_WgG
        self.integrator.integrate(domain=domain,  # Integration domain
                                  integrand=integrand,
                                  task=self.task,
                                  wd=self.wd,  # Frequency Descriptor
                                  out_wxx=out_WgG)  # Output array

        if self.hilbert:
            # The integrator only returns the spectral function and a Hilbert
            # transform is performed to return the real part of the density
            # response function.
            with self.context.timer('Hilbert transform'):
                # Make Hilbert transform
                ht = HilbertTransform(np.array(self.wd.omega_w), self.eta,
                                      timeordered=self.timeordered)
                ht(out_WgG)
            # Update the actual chi0 array
            chi0_body.data_WgG[:] += out_WgG
        chi0_body.data_WgG[:] *= prefactor

        tmp_chi0_wGG = chi0_body.copy_array_with_distribution('wGG')
        with self.context.timer('symmetrize_wGG'):
            operators = BodySymmetryOperators(symmetries, chi0_body.qpd)
            operators.symmetrize_wGG(tmp_chi0_wGG)
        chi0_body.data_WgG[:] = chi0_body.blockdist.distribute_as(
            tmp_chi0_wGG, chi0_body.nw, 'WgG')

    def construct_hermitian_task(self):
        return Hermitian(self.integrator.blockcomm, eshift=self.eshift)

    def construct_point_hilbert_task(self):
        return Hilbert(self.integrator.blockcomm, eshift=self.eshift)

    def construct_tetra_hilbert_task(self):
        return HilbertTetrahedron(self.integrator.blockcomm)

    def construct_literal_task(self):
        return GenericUpdate(
            self.eta, self.integrator.blockcomm, eshift=self.eshift)

    def print_info(self, qpd: SingleQPWDescriptor):

        if gpaw.dry_run:
```

```python
            from gpaw.mpi import SerialCommunicator
            size = gpaw.dry_run
            comm = SerialCommunicator()
            comm.size = size
        else:
            comm = self.context.comm

        q_c = qpd.q_c
        nw = len(self.wd)
        csize = comm.size
        knsize = self.integrator.kncomm.size
        bsize = self.integrator.blockcomm.size
        chisize = nw * qpd.ngmax**2 * 16. / 1024**2 / bsize

        isl = ['', f'{ctime()}',
               'Calculating chi0 body with:',
               self.get_gs_info_string(tab='    '), '',
               '    Linear response parametrization:',
               f'    q_c: [{q_c[0]}, {q_c[1]}, {q_c[2]}]',
               self.get_response_info_string(qpd, tab='    '),
               f'    comm.size: {csize}',
               f'    kncomm.size: {knsize}',
               f'    blockcomm.size: {bsize}']
        if bsize > nw:
            isl.append(
                'WARNING! Your nblocks is larger than number of frequency'
                ' points. Errors might occur, if your submodule does'
                ' not know how to handle this.')
        isl.extend(['',
                    '    Memory estimate of potentially large arrays:',
                    f'        chi0_wGG: {chisize} M / cpu',
                    '        Memory usage before allocation: '
                    f'{(maxrss() / 1024**2)} M / cpu'])
        self.context.print('\n'.join(isl))


class Chi0OpticalExtensionCalculator(Chi0ComponentPWCalculator):

    def __init__(self, *args,
                 intraband=True,
                 rate=0.0,
                 **kwargs):
        """Contruct the Chi0OpticalExtensionCalculator.

        Parameters
        ----------
        intraband : bool
            Flag for including the intraband contribution to the chi0 head.
        rate : float, str
            Phenomenological scattering rate to use in optical limit Drude term
            (in eV). If rate='eta', it uses input artificial broadening eta as
            rate. Please note that for consistency the rate is implemented as
            omegap^2 / (omega + 1j * rate)^2, which differs from some
            literature by a factor of 2.
        """
        # Serial block distribution
        super().__init__(*args, nblocks=1, **kwargs)

        # In the optical limit of metals, one must add the Drude dielectric
        # response from the free-space plasma frequency of the intraband
        # transitions to the head of chi0. This is handled by a separate
        # calculator, provided that intraband is set to True.
        if self.gs.metallic and intraband:
            from gpaw.response.chi0_drude import Chi0DrudeCalculator
            if rate == 'eta':
                rate = self.eta * Ha  # external units
            self.rate = rate
            self.drude_calc = Chi0DrudeCalculator(
                self.gs, self.context,
```

```
                qsymmetry=self.qsymmetry,
                integrationmode=self.integrationmode)
        else:
            self.drude_calc = None
            self.rate = None

    def calculate(self,
                  qpd: SingleQPWDescriptor | None = None
                  ) -> Chi0OpticalExtensionData:
        """Calculate the chi0 head and wings."""
        # Create data object
        if qpd is None:
            qpd = self.get_pw_descriptor(q_c=[0., 0., 0.])

        # wd: FrequencyDescriptor from gpaw.response.frequencies
        chi0_opt_ext = Chi0OpticalExtensionData(self.wd, qpd)

        self.print_info(qpd)

        # Define band transitions
        m1, m2 = self.get_band_transitions()

        # Perform the actual integration
        self.update_chi0_optical_extension(chi0_opt_ext, m1, m2,
                                           spins=range(self.gs.nspins))

        if self.drude_calc is not None:
            # Add intraband contribution
            # drude_calc: Chi0DrudeCalculator from gpaw.response.chi0_drude
            # chi0_drude: Chi0DrudeData from gpaw.response.chi0_data
            chi0_drude = self.drude_calc.calculate(self.wd, self.rate)
            chi0_opt_ext.head_Wvv[:] += chi0_drude.chi_Zvv

        return chi0_opt_ext

    def update_chi0_optical_extension(
            self,
            chi0_optical_extension: Chi0OpticalExtensionData,
            m1: int, m2: int,
            spins: list | range):
        """In-place calculation of the chi0 head and wings.

        Parameters
        ----------
        m1 : int
            Lower band cutoff for band summation
        m2 : int
            Upper band cutoff for band summation
        spins : list
            List of spin indices to include in the calculation
        """
        self.context.print('Integrating chi0 head and wings.')
        chi0_opt_ext = chi0_optical_extension
        qpd = chi0_opt_ext.qpd

        symmetries, generator, domain, prefactor = self.get_integration_domain(
            qpd.q_c, spins)
        integrand = Chi0Integrand(self, qpd=qpd, generator=generator,
                                  optical=True, m1=m1, m2=m2)

        # We integrate the head and wings together, using the combined index P
        # index v = (x, y, z)
        # index G = (G0, G1, G2, ...)
        # index P = (x, y, z, G1, G2, ...)
        WxvP_shape = list(chi0_opt_ext.WxvG_shape)
        WxvP_shape[-1] += 2
        tmp_chi0_WxvP = np.zeros(WxvP_shape, complex)
        self.integrator.integrate(domain=domain,  # Integration domain
                                  integrand=integrand,
```

```python
                                 task=self.task,
                                 wd=self.wd,  # Frequency Descriptor
                                 out_wxx=tmp_chi0_WxvP)  # Output array
        if self.hilbert:
            with self.context.timer('Hilbert transform'):
                ht = HilbertTransform(np.array(self.wd.omega_w), self.eta,
                                      timeordered=self.timeordered)
                ht(tmp_chi0_WxvP)
        tmp_chi0_WxvP *= prefactor

        # Fill in wings part of the data, but leave out the head part (G0)
        chi0_opt_ext.wings_WxvG[..., 1:] += tmp_chi0_WxvP[..., 3:]
        # Fill in the head
        chi0_opt_ext.head_Wvv[:] += tmp_chi0_WxvP[:, 0, :3, :3]
        # Symmetrize
        operators = WingSymmetryOperators(symmetries, qpd)
        operators.symmetrize_wxvG(chi0_opt_ext.wings_WxvG)
        operators.symmetrize_wvv(chi0_opt_ext.head_Wvv)

    def construct_hermitian_task(self):
        return HermitianOpticalLimit()

    def construct_point_hilbert_task(self):
        return HilbertOpticalLimit()

    def construct_tetra_hilbert_task(self):
        return HilbertOpticalLimitTetrahedron()

    def construct_literal_task(self):
        return OpticalLimit(eta=self.eta)

    def print_info(self, qpd: SingleQPWDescriptor):
        """Print information about optical extension calculation."""
        isl = ['',
               f'{ctime()}',
               'Calculating chi0 optical extensions with:',
               self.get_gs_info_string(tab='    '),
               '',
               '    Linear response parametrization:',
               self.get_response_info_string(qpd, tab='    ')]
        self.context.print('\n'.join(isl))


def get_frequency_descriptor(
        frequencies: ArrayLike1D | dict[str, Any] | None = None, *,
        gs: ResponseGroundStateAdapter | None = None,
        nbands: int | None = None):
    """Helper function to generate frequency descriptors.

    In most cases, the `frequencies` input can be processed directly via
    wd = FrequencyDescriptor.from_array_or_dict(frequencies),
    but in cases where `frequencies` does not specify omegamax, it is
    calculated from the input ground state adapter.
    """
    if frequencies is None:
        frequencies = {'type': 'nonlinear'}  # default frequency grid
    if isinstance(frequencies, dict) and frequencies.get('omegamax') is None:
        assert gs is not None
        frequencies['omegamax'] = get_omegamax(gs, nbands)
    return FrequencyDescriptor.from_array_or_dict(frequencies)


def get_omegamax(gs: ResponseGroundStateAdapter,
                 nbands: int | None = None):
    """Get the maxmimum eigenvalue difference including nbands, in eV."""
    epsmin, epsmax = gs.get_eigenvalue_range(nbands=nbands)
    return (epsmax - epsmin) * Ha
```