```python
from __future__ import annotations
from abc import ABC, abstractmethod
from dataclasses import dataclass
import sys

import numpy as np
from ase.units import Hartree

import gpaw.mpi as mpi

from gpaw.response.pw_parallelization import Blocks1D
from gpaw.response.coulomb_kernels import CoulombKernel
from gpaw.response.dyson import DysonEquation
from gpaw.response.density_kernels import DensityXCKernel
from gpaw.response.chi0 import Chi0Calculator, get_frequency_descriptor
from gpaw.response.chi0_data import Chi0Data
from gpaw.response.pair import get_gs_and_context

from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from gpaw.response.groundstate import CellDescriptor
    from gpaw.response.frequencies import FrequencyDescriptor
    from gpaw.response.pair_functions import SingleQPWDescriptor


"""
On the notation in this module.

When calculating properties such as the dielectric function, EELS spectrum and
polarizability there are many inherent subtleties relating to (ir)reducible
representations and inclusion of local-field effects. For the reciprocal space
representation of the Coulomb potential, we use the following notation

v or v(q): The bare Coulomb interaction, 4π/|G+q|²

V or V(q): The specified Coulomb interaction. Will usually be either the bare
           interaction or a truncated version hereof.

V̄ or V̄(q): The modified Coulomb interaction. Equal to V(q) for finite
           reciprocal wave vectors G > 0, but modified to exclude long-range
           interactions, that is, equal to 0 for G = 0.
"""


@dataclass
class Chi0DysonEquations:
    chi0: Chi0Data
    coulomb: CoulombKernel
    xc_kernel: DensityXCKernel | None
    cd: CellDescriptor

    def __post_init__(self):
        if self.coulomb.truncation is None:
            self.bare_coulomb = self.coulomb
        else:
            self.bare_coulomb = self.coulomb.new(truncation=None)
        # When inverting the Dyson equation, we distribute frequencies globally
        blockdist = self.chi0.body.blockdist.new_distributor(nblocks='max')
        self.wblocks = Blocks1D(blockdist.blockcomm, len(self.chi0.wd))

    @staticmethod
    def _normalize(direction):
        if isinstance(direction, str):
            d_v = {'x': [1, 0, 0],
                   'y': [0, 1, 0],
                   'z': [0, 0, 1]}[direction]
        else:
            d_v = direction
```

```
        d_v = np.asarray(d_v) / np.linalg.norm(d_v)
        return d_v

    def get_chi0_wGG(self, direction='x'):
        chi0 = self.chi0
        chi0_wGG = chi0.body.get_distributed_frequencies_array().copy()
        if chi0.qpd.optical_limit:
            # Project head and wings along the input direction
            d_v = self._normalize(direction)
            W_w = self.wblocks.myslice
            chi0_wGG[:, 0] = np.dot(d_v, chi0.chi0_WxvG[W_w, 0])
            chi0_wGG[:, :, 0] = np.dot(d_v, chi0.chi0_WxvG[W_w, 1])
            chi0_wGG[:, 0, 0] = np.dot(d_v, np.dot(chi0.chi0_Wvv[W_w], d_v).T)
        return chi0_wGG

    def get_coulomb_scaled_kernel(self, modified=False, Kxc_GG=None):
        """Get the Hxc kernel rescaled by the bare Coulomb potential v(q).

        Calculates

        K̃(q) = v^(-1/2)(q) K_Hxc(q) v^(-1/2)(q),

        where v(q) is the bare Coulomb potential and

        K_Hxc(q) = V(q) + K_xc(q).

        When using the `modified` flag, the specified Coulomb kernel will be
        replaced with its modified analogue,

        V(q) -> V̄(q)
        """
        qpd = self.chi0.qpd
        if self.coulomb is self.bare_coulomb:
            v_G = self.coulomb.V(qpd)  # bare Coulomb interaction
            K_GG = np.eye(len(v_G), dtype=complex)
        else:
            v_G = self.bare_coulomb.V(qpd)
            V_G = self.coulomb.V(qpd)
            K_GG = np.diag(V_G / v_G)
        if modified:
            K_GG[0, 0] = 0.
        if Kxc_GG is not None:
            sqrtv_G = v_G**0.5
            K_GG += Kxc_GG / sqrtv_G / sqrtv_G[:, np.newaxis]
        return v_G, K_GG

    @staticmethod
    def invert_dyson_like_equation(in_wGG, K_GG, reuse_buffer=True):
        """Generalized Dyson equation invertion.

        Calculates

        B(q,ω) = [1 - A(q,ω) K(q)]⁻¹ A(q,ω)

        while possibly storing the output B(q,ω) in the input A(q,ω) buffer.
        """
        if reuse_buffer:
            out_wGG = in_wGG
        else:
            out_wGG = np.zeros_like(in_wGG)
        for w, in_GG in enumerate(in_wGG):
            out_wGG[w] = DysonEquation(in_GG, in_GG @ K_GG).invert()
        return out_wGG

    def rpa_density_response(self, direction='x', qinf_v=None):
        """Calculate the RPA susceptibility for (semi-)finite q.

        Currently this is only used by the QEH code, why we don't support a top
        level user interface.
```

```
    """
    # Extract χ₀(q,ω)
    qpd = self.chi0.qpd
    chi0_wGG = self.get_chi0_wGG(direction=direction)
    if qpd.optical_limit:
        # Restore the q-dependence of the head and wings in the q→0 limit
        assert qinf_v is not None and np.linalg.norm(qinf_v) > 0.
        d_v = self._normalize(direction)
        chi0_wGG[:, 1:, 0] *= np.dot(qinf_v, d_v)
        chi0_wGG[:, 0, 1:] *= np.dot(qinf_v, d_v)
        chi0_wGG[:, 0, 0] *= np.dot(qinf_v, d_v)**2
    # Invert Dyson equation, χ(q,ω) = [1 - χ₀(q,ω) V(q)]⁻¹ χ₀(q,ω)
    V_GG = self.coulomb.kernel(qpd, q_v=qinf_v)
    chi_wGG = self.invert_dyson_like_equation(chi0_wGG, V_GG)
    return qpd, chi_wGG, self.wblocks

def inverse_dielectric_function(self, direction='x'):
    """Calculate v^(1/2) χ v^(1/2), from which ε⁻¹(q,ω) is constructed."""
    return InverseDielectricFunction.from_chi0_dyson_eqs(
        self, *self.calculate_vchi_symm(direction=direction))

def calculate_vchi_symm(self, direction='x', modified=False):
    """Calculate v^(1/2) χ v^(1/2).

    Starting from the TDDFT Dyson equation

    χ(q,ω) = χ₀(q,ω) + χ₀(q,ω) K_Hxc(q,ω) χ(q,ω),                (1)

    the Coulomb scaled susceptibility,

    χ̃(q,ω) = v^(1/2)(q) χ(q,ω) v^(1/2)(q)

    can be calculated from the Dyson-like equation

    χ̃(q,ω) = χ̃₀(q,ω) + χ̃₀(q,ω) K̃(q,ω) χ̃(q,ω)                    (2)

    where

    K̃(q,ω) = v^(-1/2)(q) K_Hxc(q,ω) v^(-1/2)(q).

    Here v(q) refers to the bare Coulomb potential. It should be emphasized
    that invertion of (2) rather than (1) is not merely a rescaling
    excercise. In the optical q → 0 limit, the Coulomb kernel v(q) diverges
    as 1/|G+q|² while the Kohn-Sham susceptibility χ₀(q,ω) vanishes as
    |G+q|². Treating v^(1/2)(q) χ₀(q,ω) v^(1/2)(q) as a single variable,
    the effects of this cancellation can be treated accurately within k.p
    perturbation theory.
    """
    chi0_wGG = self.get_chi0_wGG(direction=direction)
    Kxc_GG = self.xc_kernel(self.chi0.qpd, chi0_wGG=chi0_wGG) \
        if self.xc_kernel else None
    v_G, K_GG = self.get_coulomb_scaled_kernel(
        modified=modified, Kxc_GG=Kxc_GG)
    # Calculate v^(1/2)(q) χ₀(q,ω) v^(1/2)(q)
    sqrtv_G = v_G**0.5
    vchi0_symm_wGG = chi0_wGG  # reuse buffer
    for w, chi0_GG in enumerate(chi0_wGG):
        vchi0_symm_wGG[w] = chi0_GG * sqrtv_G * sqrtv_G[:, np.newaxis]
    # Invert Dyson equation
    vchi_symm_wGG = self.invert_dyson_like_equation(
        vchi0_symm_wGG, K_GG, reuse_buffer=False)
    return vchi0_symm_wGG, vchi_symm_wGG

def customized_dielectric_function(self, direction='x'):
    """Calculate E(q,ω) = 1 - V(q) P(q,ω)."""
    V_GG = self.coulomb.kernel(self.chi0.qpd)
    P_wGG = self.polarizability_operator(direction=direction)
    nG = len(V_GG)
    eps_wGG = P_wGG  # reuse buffer
```

```
        for w, P_GG in enumerate(P_wGG):
            eps_wGG[w] = np.eye(nG) - V_GG @ P_GG
        return CustomizableDielectricFunction.from_chi0_dyson_eqs(
            self, eps_wGG)

    def bare_dielectric_function(self, direction='x'):
        """Calculate v^(1/2) χ̄ v^(1/2), from which ε̄=1-v χ̄ is constructed.

        The unscreened susceptibility is given by the Dyson-like equation

        χ̄(q,ω) = P(q,ω) + P(q,ω) V̄(q) χ̄(q,ω),                        (3)

        In the special case of RPA, where P(q,ω) = χ₀(q,ω), one may notice that
        the Dyson-like equation (3) is exactly identical to the TDDFT Dyson
        equation (1) when replacing the Hartree-exchange-correlation kernel
        with the modified Coulomb interaction:

        K_Hxc(q) -> V̄(q).

        We may thus reuse that functionality to calculate v^(1/2) χ̄ v^(1/2).
        """
        if self.xc_kernel:
            raise NotImplementedError(
                'Calculation of the bare dielectric function within TDDFT has '
                'not yet been implemented. For TDDFT dielectric properties, '
                'please calculate the inverse dielectric function.')
        vP_symm_wGG, vchibar_symm_wGG = self.calculate_vchi_symm(
            direction=direction, modified=True)
        return BareDielectricFunction.from_chi0_dyson_eqs(
            self, vP_symm_wGG, vchibar_symm_wGG)

    def polarizability_operator(self, direction='x'):
        """Calculate the polarizability operator P(q,ω).

        Depending on the theory (RPA, TDDFT, MBPT etc.), the polarizability
        operator is approximated in various ways see e.g.
        [Rev. Mod. Phys. 74, 601 (2002)].

        In RPA:
            P(q,ω) = χ₀(q,ω)

        In TDDFT:
            P(q,ω) = [1 - χ₀(q,ω) K_xc(q,ω)]⁻¹ χ₀(q,ω)
        """
        chi0_wGG = self.get_chi0_wGG(direction=direction)
        if not self.xc_kernel:  # RPA
            return chi0_wGG
        # TDDFT (in adiabatic approximations to the kernel)
        if self.chi0.qpd.optical_limit:
            raise NotImplementedError(
                'Calculation of the TDDFT dielectric function via the '
                'polarizability operator has not been implemented for the '
                'optical limit. Please calculate the inverse dielectric '
                'function instead.')
            # The TDDFT implementation here is invalid in the optical limit
            # since the chi0_wGG variable already contains Coulomb effects,
            # chi0_wGG ~ V^(1/2) χ₀ V^(1/2)/(4π).
            # Furthermore, a direct evaluation of V(q) P(q,ω) does not seem
            # sensible, since it does not account for the exact cancellation
            # of the q-dependences of the two functions.
            # In principle, one could treat the v(q) P(q,ω) product in
            # perturbation theory, similar to the χ₀(q,ω) v(q) product in the
            # Dyson equation for χ, but unless we need to calculate the TDDFT
            # polarizability using truncated kernels, this isn't really
            # necessary.
        Kxc_GG = self.xc_kernel(self.chi0.qpd, chi0_wGG=chi0_wGG)
        return self.invert_dyson_like_equation(chi0_wGG, Kxc_GG)
```

```
@dataclass
class DielectricFunctionData(ABC):
    cd: CellDescriptor
    qpd: SingleQPWDescriptor
    wd: FrequencyDescriptor
    wblocks: Blocks1D
    coulomb: CoulombKernel
    bare_coulomb: CoulombKernel

    @classmethod
    def from_chi0_dyson_eqs(cls, chi0_dyson_eqs, *args, **kwargs):
        chi0 = chi0_dyson_eqs.chi0
        return cls(chi0_dyson_eqs.cd, chi0.qpd,
                   chi0.wd, chi0_dyson_eqs.wblocks,
                   chi0_dyson_eqs.coulomb, chi0_dyson_eqs.bare_coulomb,
                   *args, **kwargs)

    def _macroscopic_component(self, in_wGG):
        return self.wblocks.all_gather(in_wGG[:, 0, 0])

    @property
    def v_G(self):
        return self.bare_coulomb.V(self.qpd)

    @abstractmethod
    def macroscopic_dielectric_function(self) -> ScalarResponseFunctionSet:
        """Get the macroscopic dielectric function ε_M(q,ω)."""

    def dielectric_constant(self):
        return self.macroscopic_dielectric_function().static_limit.real

    def eels_spectrum(self):
        """Get the macroscopic EELS spectrum.

        The spectrum is defined as

                                              1
        EELS(q,ω) ≡ -Im ε⁻¹(q,ω) = -Im ──────────.
                         00           ε (q,ω)
                                       M

        In addition to the many-body spectrum, we also calculate the
        EELS spectrum in the relevant independent-particle approximation,
        here defined as

                            1
        EELS₀(ω) = -Im ──────────.
                        IP
                       ε (q,ω)
                        M
        """
        _, eps0_W, eps_W = self.macroscopic_dielectric_function().arrays
        eels0_W = -(1. / eps0_W).imag
        eels_W = -(1. / eps_W).imag
        return ScalarResponseFunctionSet(self.wd, eels0_W, eels_W)

    def polarizability(self):
        """Get the macroscopic polarizability α_M(q,ω).

        In the special case where dielectric properties are calculated
        solely based on the bare Coulomb interaction V(q) = v(q), the
        macroscopic part of the electronic polarizability α(q,ω) is related
        directly to the macroscopic dielectric function ε_M(q,ω).

        In particular, we define the macroscroscopic polarizability so as to
        make it independent of the nonperiodic cell vector lengths. Namely,

        α (q,ω) ≡ Λ α (q,ω)
         M          00
```

```
        where Λ is the nonperiodic hypervolume of the unit cell. Thus,

        α_M(q,ω) = Λ/(4π) (ϵ_M(q,ω) - 1) = Λ/(4π) (ε_M(q,ω) - 1),

        where the latter equality only holds in the special case V(q) = v(q).

        In addition to α_M(q,ω), we calculate also the macroscopic
        polarizability in the relevant independent-particle approximation by
        replacing ϵ_M with ε_M^(IP).
        """
        if self.coulomb is not self.bare_coulomb:
            raise ValueError(
                'When using a truncated Coulomb kernel, the polarizability '
                'cannot be calculated based on the macroscopic dielectric '
                'function. Please calculate the bare dielectric function '
                'instead.')
        _, eps0_W, eps_W = self.macroscopic_dielectric_function().arrays
        return self._polarizability(eps0_W, eps_W)

    def _polarizability(self, eps0_W, eps_W):
        L = self.cd.nonperiodic_hypervolume
        alpha0_W = L / (4 * np.pi) * (eps0_W - 1)
        alpha_W = L / (4 * np.pi) * (eps_W - 1)
        return ScalarResponseFunctionSet(self.wd, alpha0_W, alpha_W)


@dataclass
class InverseDielectricFunction(DielectricFunctionData):
    """Data class for the inverse dielectric function ε⁻¹(q,ω).

    The inverse dielectric function characterizes the longitudinal response

    V (q,ω) = ε⁻¹(q,ω) V (q,ω),
     tot               ext

    where the induced potential due to the electronic system is given by vχ,

    ε⁻¹(q,ω) = 1 + v(q) χ(q,ω).

    In this data class, ε⁻¹ is cast in terms if its symmetrized representation

    ε̃⁻¹(q,ω) = v^(-1/2)(q) ε⁻¹(q,ω) v^(1/2)(q),

    that is, in terms of v^(1/2)(q) χ(q,ω) v^(1/2)(q).

    Please remark that v(q) here refers to the bare Coulomb potential
    irregardless of whether χ(q,ω) was determined using a truncated analogue.
    """
    vchi0_symm_wGG: np.ndarray  # v^(1/2)(q) χ₀(q,ω) v^(1/2)(q)
    vchi_symm_wGG: np.ndarray

    def macroscopic_components(self):
        vchi0_W = self._macroscopic_component(self.vchi0_symm_wGG)
        vchi_W = self._macroscopic_component(self.vchi_symm_wGG)
        return vchi0_W, vchi_W

    def macroscopic_dielectric_function(self):
        """Get the macroscopic dielectric function ε_M(q,ω).

        Calculates

                        1
        ε (q,ω) = ──────────
         M         ε⁻¹(q,ω)
                    00

        along with the macroscopic dielectric function in the independent-
        particle random-phase approximation [Rev. Mod. Phys. 74, 601 (2002)],
```

```
         IPRPA
        ε (q,ω) = 1 - v(q) χ⁰(q,ω)
         M                 00

        that is, neglecting local-field and exchange-correlation effects.
        """
        vchi0_W, vchi_W = self.macroscopic_components()
        eps0_W = 1 - vchi0_W
        eps_W = 1 / (1 + vchi_W)
        return ScalarResponseFunctionSet(self.wd, eps0_W, eps_W)

    def dynamic_susceptibility(self):
        """Get the macroscopic components of χ(q,ω) and χ₀(q,ω)."""
        vchi0_W, vchi_W = self.macroscopic_components()
        v0 = self.v_G[0]  # Macroscopic Coulomb potential (4π/q²)
        return ScalarResponseFunctionSet(self.wd, vchi0_W / v0, vchi_W / v0)


@dataclass
class CustomizableDielectricFunction(DielectricFunctionData):
    """Data class for customized dielectric functions E(q,ω).

    E(q,ω) is customizable in the sense that bare Coulomb interaction v(q) is
    replaced by an arbitrary interaction V(q) in the formula for ε(q,ω),

    E(q,ω) = 1 - V(q) P(q,ω),

    where P is the polarizability operator [Rev. Mod. Phys. 74, 601 (2002)].
    Thus, for any truncated or otherwise cusomized interaction V(q) ≠ v(q),
    E(q,ω) ≠ ε(q,ω) and E⁻¹(q,ω) ≠ ε⁻¹(q,ω).
    """
    eps_wGG: np.ndarray

    def macroscopic_customized_dielectric_function(self):
        """Get the macroscopic customized dielectric function E_M(q,ω).

        We define the macroscopic customized dielectric function as

                     1
        E (q,ω) = ──────────,
         M         E⁻¹(q,ω)
                    00

        and calculate also the macroscopic dielectric function in the
        customizable independent-particle approximation:

         cIP
        ε (q,ω) = 1 - VP (q,ω) = E (q,ω).
         M             00        00
        """
        eps0_W = self._macroscopic_component(self.eps_wGG)
        # Invert E(q,ω) one frequency at a time to compute E_M(q,ω)
        eps_w = np.zeros((self.wblocks.nlocal,), complex)
        for w, eps_GG in enumerate(self.eps_wGG):
            eps_w[w] = 1 / np.linalg.inv(eps_GG)[0, 0]
        eps_W = self.wblocks.all_gather(eps_w)
        return ScalarResponseFunctionSet(self.wd, eps0_W, eps_W)

    def macroscopic_dielectric_function(self):
        """Get the macroscopic dielectric function ε_M(q,ω).

        In the special case V(q) = v(q), E_M(q,ω) = ε_M(q,ω).
        """
        if self.coulomb is not self.bare_coulomb:
            raise ValueError(
                'The macroscopic dielectric function is defined in terms of '
                'the bare Coulomb interaction. To truncate the Hartree '
                'electron-electron correlations, please calculate the inverse '
```

```
                'dielectric function instead.')
        return self.macroscopic_customized_dielectric_function()


@dataclass
class BareDielectricFunction(DielectricFunctionData):
    """Data class for the bare (unscreened) dielectric function.

    The bare dielectric function is defined in terms of the unscreened
    susceptibility,

    -
    ϵ(q,ω) = 1 - v(q) χ(q,ω),

    here represented in terms of v^(1/2)(q) χ(q,ω) v^(1/2)(q).
    """
    vP_symm_wGG: np.ndarray  # v^(1/2) P v^(1/2)
    vchibar_symm_wGG: np.ndarray

    def macroscopic_components(self):
        vP_W = self._macroscopic_component(self.vP_symm_wGG)
        vchibar_W = self._macroscopic_component(self.vchibar_symm_wGG)
        return vP_W, vchibar_W

    def macroscopic_bare_dielectric_function(self):
        """Get the macroscopic bare dielectric function ϵ_M(q,ω).

        Calculates

                -
        ϵ_M(q,ω) = ϵ (q,ω)
                    00

        along with the macroscopic dielectric function in the independent-
        particle approximation:

         IP
        ε (q,ω) = 1 - vP (q,ω).
         M             00
        """
        vP_W, vchibar_W = self.macroscopic_components()
        eps0_W = 1. - vP_W
        eps_W = 1. - vchibar_W
        return ScalarResponseFunctionSet(self.wd, eps0_W, eps_W)

    def macroscopic_dielectric_function(self):
        """Get the macroscopic dielectric function ε_M(q,ω).

        In the special case where the unscreened susceptibility is calculated
        using the bare Coulomb interaction,

        -
        χ(q,ω) = P(q,ω) + P(q,ω) v(q) χ(q,ω),

        it holds identically that [Rev. Mod. Phys. 74, 601 (2002)]:

        ε_M(q,ω) = ϵ_M(q,ω).
        """
        if self.coulomb is not self.bare_coulomb:
            raise ValueError(
                'The macroscopic dielectric function cannot be obtained from '
                'the bare dielectric function calculated based on a truncated '
                'Coulomb interaction. Please calculate the inverse dielectric '
                'function instead')
        return self.macroscopic_bare_dielectric_function()

    def polarizability(self):
        """Get the macroscopic polarizability α_M(q,ω).

        In the most general case, the electronic polarizability can be defined
        in terms of the unscreened susceptibility
                                -               -
```

```
        α(q,ω) ≡ - 1/(4π) v(q) χ(q,ω) = 1/(4π) (ϵ(q,ω) - 1).

        This is especially useful in systems of reduced dimensionality, where
        χ(q,ω) needs to be calculated using a truncated Coulomb kernel in order
        to achieve convergence in a feasible way.
        """
        _, eps0_W, eps_W = self.macroscopic_bare_dielectric_function().arrays
        return self._polarizability(eps0_W, eps_W)


class DielectricFunctionCalculator:
    def __init__(self, chi0calc: Chi0Calculator):
        self.chi0calc = chi0calc
        self.gs = chi0calc.gs
        self.context = chi0calc.context

        self._chi0cache: dict[tuple[str, ...], Chi0Data] = {}

    def get_chi0(self, q_c: list | np.ndarray) -> Chi0Data:
        """Get the Kohn-Sham susceptibility χ₀(q,ω) for input wave vector q.

        Keeps a cache of χ₀ for the latest calculated wave vector, thus
        allowing for investigation of multiple dielectric properties,
        Coulomb truncations, xc kernels etc. without recalculating χ₀.
        """
        # As cache key, we round off and use a string representation.
        # Not very elegant, but it should work almost always.
        q_key = [f'{q:.10f}' for q in q_c]
        key = tuple(q_key)
        if key not in self._chi0cache:
            self._chi0cache.clear()
            self._chi0cache[key] = self.chi0calc.calculate(q_c)
            self.context.write_timer()
        return self._chi0cache[key]

    def get_chi0_dyson_eqs(self,
                           q_c: list | np.ndarray = [0, 0, 0],
                           truncation: str | None = None,
                           xc: str = 'RPA',
                           **xckwargs
                           ) -> Chi0DysonEquations:
        """Set up the Dyson equation for χ(q,ω) at given wave vector q.

        Parameters
        ----------
        truncation : str or None
            Truncation of the Hartree kernel.
        xc : str
            Exchange-correlation kernel for LR-TDDFT calculations.
            If xc == 'RPA', the dielectric response is treated in the random
            phase approximation.
        **xckwargs
            Additional parameters for the chosen xc kernel.
        """
        chi0 = self.get_chi0(q_c)
        coulomb = CoulombKernel.from_gs(self.gs, truncation=truncation)
        if xc == 'RPA':
            xc_kernel = None
        else:
            xc_kernel = DensityXCKernel.from_functional(
                self.gs, self.context, functional=xc, **xckwargs)
        return Chi0DysonEquations(chi0, coulomb, xc_kernel, self.gs.cd)

    def get_bare_dielectric_function(self, direction='x', **kwargs
                                     ) -> BareDielectricFunction:
        """Calculate the bare dielectric function ε̄(q,ω) = 1 - v(q) χ̄(q,ω).

        Here v(q) is the bare Coulomb potential while χ̄ is the unscreened
        susceptibility calculated based on the modified (and possibly
```

```
        truncated) Coulomb potential.
        """
        return self.get_chi0_dyson_eqs(
            **kwargs).bare_dielectric_function(direction=direction)

    def get_literal_dielectric_function(self, direction='x', **kwargs
                                        ) -> CustomizableDielectricFunction:
        """Calculate the dielectric function ε(q,ω) = 1 - v(q) P(q,ω)."""
        return self.get_chi0_dyson_eqs(
            truncation=None, **kwargs).customized_dielectric_function(
                direction=direction)

    def get_customized_dielectric_function(self, direction='x', *,
                                           truncation: str, **kwargs
                                           ) -> CustomizableDielectricFunction:
        """Calculate the customized dielectric function E(q,ω) = 1 -V(q)P(q,ω).

        In comparison with the literal dielectric function, the bare Coulomb
        interaction has here been replaced with a truncated analogue v(q)→V(q).
        """
        return self.get_chi0_dyson_eqs(
            truncation=truncation, **kwargs).customized_dielectric_function(
                direction=direction)

    def get_inverse_dielectric_function(self, direction='x', **kwargs
                                        ) -> InverseDielectricFunction:
        """Calculate the inverse dielectric function ε⁻¹(q,ω) = v(q) χ(q,ω).
        """
        return self.get_chi0_dyson_eqs(
            **kwargs).inverse_dielectric_function(direction=direction)


class DielectricFunction(DielectricFunctionCalculator):
    """This class defines dielectric function related physical quantities."""

    def __init__(self, calc, *,
                 frequencies=None,
                 ecut=50,
                 hilbert=True,
                 nbands=None, eta=0.2,
                 intraband=True, nblocks=1, world=mpi.world, txt=sys.stdout,
                 truncation=None,
                 qsymmetry=True,
                 integrationmode=None, rate=0.0,
                 eshift: float | None = None):
        """Creates a DielectricFunction object.

        calc: str
            The ground-state calculation file that the linear response
            calculation is based on.
        frequencies:
            Input parameters for frequency_grid.
            Can be an array of frequencies to evaluate the response function at
            or dictionary of parameters for build-in nonlinear grid
            (see :ref:`frequency grid`).
        ecut: float
            Plane-wave cut-off.
        hilbert: bool
            Use hilbert transform.
        nbands: int
            Number of bands from calculation.
        eta: float
            Broadening parameter.
        intraband: bool
            Include intraband transitions.
        world: comm
            mpi communicator.
        nblocks: int
            Split matrices in nblocks blocks and distribute them G-vectors or
```

```
        frequencies over processes.
    txt: str
        Output file.
    truncation: str or None
        None for no truncation.
        '2D' for standard analytical truncation scheme.
        Non-periodic directions are determined from k-point grid
    eshift: float
        Shift unoccupied bands
    """
    gs, context = get_gs_and_context(calc, txt, world, timer=None)
    wd = get_frequency_descriptor(frequencies, gs=gs, nbands=nbands)

    chi0calc = Chi0Calculator(
        gs, context, nblocks=nblocks,
        wd=wd,
        ecut=ecut, nbands=nbands, eta=eta,
        hilbert=hilbert,
        intraband=intraband,
        qsymmetry=qsymmetry,
        integrationmode=integrationmode,
        rate=rate, eshift=eshift
    )
    super().__init__(chi0calc)
    self.truncation = truncation

def get_frequencies(self) -> np.ndarray:
    """Return frequencies (in eV) that the χ is evaluated on."""
    return self.chi0calc.wd.omega_w * Hartree

def get_dynamic_susceptibility(self, *args, xc='ALDA',
                               filename='chiM_w.csv',
                               **kwargs):
    dynsus = self.get_inverse_dielectric_function(
        *args, xc=xc, truncation=self.truncation,
        **kwargs).dynamic_susceptibility()
    if filename:
        dynsus.write(filename)
    return dynsus.unpack()

def get_dielectric_function(self, *args, filename='df.csv', **kwargs):
    """Calculate the dielectric function.

    Generates a file 'df.csv', unless filename is set to None.

    Returns
    -------
    df_NLFC_w: np.ndarray
        Dielectric function without local field corrections.
    df_LFC_w: np.ndarray
        Dielectric functio with local field corrections.
    """
    df = self.get_inverse_dielectric_function(
        *args, truncation=self.truncation,
        **kwargs).macroscopic_dielectric_function()
    if filename:
        df.write(filename)
    return df.unpack()

def get_eels_spectrum(self, *args, filename='eels.csv', **kwargs):
    """Calculate the macroscopic EELS spectrum.

    Generates a file 'eels.csv', unless filename is set to None.

    Returns
    -------
    eels0_w: np.ndarray
        Spectrum in the independent-particle random-phase approximation.
    eels_w: np.ndarray
```

```
            Fully screened EELS spectrum.
        """
        eels = self.get_inverse_dielectric_function(
            *args, truncation=self.truncation, **kwargs).eels_spectrum()
        if filename:
            eels.write(filename)
        return eels.unpack()

    def get_polarizability(self, q_c: list | np.ndarray = [0, 0, 0],
                           direction='x', filename='polarizability.csv',
                           **xckwargs):
        """Calculate the macroscopic polarizability.

        Generate a file 'polarizability.csv', unless filename is set to None.

        Returns:
        --------
        alpha0_w: np.ndarray
            Polarizability calculated without local-field corrections
        alpha_w: np.ndarray
            Polarizability calculated with local-field corrections.
        """
        chi0_dyson_eqs = self.get_chi0_dyson_eqs(
            q_c, truncation=self.truncation, **xckwargs)
        if self.truncation:
            # eps: BareDielectricFunction
            method = chi0_dyson_eqs.bare_dielectric_function
        else:
            # eps: CustomizableDielectricFunction
            method = chi0_dyson_eqs.customized_dielectric_function
        eps = method(direction=direction)
        pol = eps.polarizability()
        if filename:
            pol.write(filename)
        return pol.unpack()

    def get_macroscopic_dielectric_constant(self, xc='RPA', direction='x'):
        """Calculate the macroscopic dielectric constant.

        The macroscopic dielectric constant is defined as the real part of the
        dielectric function in the static limit.

        Returns:
        --------
        eps0: float
            Dielectric constant without local field corrections.
        eps: float
            Dielectric constant with local field correction. (RPA, ALDA)
        """
        return self.get_inverse_dielectric_function(
            xc=xc, direction=direction).dielectric_constant()


# ----- Serialized dataclasses and IO ----- #


@dataclass
class ScalarResponseFunctionSet:
    """A set of scalar response functions rf₀(ω) and rf(ω)."""
    wd: FrequencyDescriptor
    rf0_w: np.ndarray
    rf_w: np.ndarray

    @property
    def arrays(self):
        return self.wd.omega_w * Hartree, self.rf0_w, self.rf_w

    def unpack(self):
        # Legacy feature to support old DielectricFunction output format
```

```
        # ... to be deprecated ...
        return self.rf0_w, self.rf_w

    def write(self, filename):
        if mpi.rank == 0:
            write_response_function(filename, *self.arrays)

    @property
    def static_limit(self):
        """Return the value of the response functions in the static limit."""
        w0 = np.argmin(np.abs(self.wd.omega_w))
        assert abs(self.wd.omega_w[w0]) < 1e-8
        return np.array([self.rf0_w[w0], self.rf_w[w0]])


def write_response_function(filename, omega_w, rf0_w, rf_w):
    with open(filename, 'w') as fd:
        for omega, rf0, rf in zip(omega_w, rf0_w, rf_w):
            if rf0_w.dtype == complex:
                print('%.6f, %.6f, %.6f, %.6f, %.6f' %
                      (omega, rf0.real, rf0.imag, rf.real, rf.imag),
                      file=fd)
            else:
                print(f'{omega:.6f}, {rf0:.6f}, {rf:.6f}', file=fd)


def read_response_function(filename):
    """Read a stored response function file"""
    d = np.loadtxt(filename, delimiter=',')
    omega_w = np.array(d[:, 0], float)

    if d.shape[1] == 3:
        # Real response function
        rf0_w = np.array(d[:, 1], float)
        rf_w = np.array(d[:, 2], float)
    elif d.shape[1] == 5:
        rf0_w = np.array(d[:, 1], complex)
        rf0_w.imag = d[:, 2]
        rf_w = np.array(d[:, 3], complex)
        rf_w.imag = d[:, 4]
    else:
        raise ValueError(f'Unexpected array dimension {d.shape}')

    return omega_w, rf0_w, rf_w
```