```python
from __future__ import annotations
import pickle
import warnings
from math import pi, isclose
from pathlib import Path
from collections.abc import Iterable

import numpy as np

from ase.parallel import paropen
from ase.units import Ha

from gpaw import GPAW, debug
import gpaw.mpi as mpi
from gpaw.hybrids.eigenvalues import non_self_consistent_eigenvalues
from gpaw.pw.descriptor import (count_reciprocal_vectors, PWMapping)
from gpaw.utilities.progressbar import ProgressBar

from gpaw.response import ResponseContext, ResponseGroundStateAdapter
from gpaw.response.chi0 import Chi0Calculator, get_frequency_descriptor
from gpaw.response.pair import phase_shifted_fft_indices
from gpaw.response.pair_functions import SingleQPWDescriptor
from gpaw.response.pw_parallelization import Blocks1D
from gpaw.response.screened_interaction import (initialize_w_calculator,
                                                GammaIntegrationMode)
from gpaw.response.coulomb_kernels import CoulombKernel
from gpaw.response import timer
from gpaw.response.mpa_sampling import mpa_frequency_sampling
from gpaw.mpi import broadcast_exception

from ase.utils.filecache import MultiFileJSONCache as FileCache
from contextlib import ExitStack
from ase.parallel import broadcast


def compare_inputs(inp1, inp2, rel_tol=1e-14, abs_tol=1e-14):
    """
    Compares nested structures of dictionarys, lists, etc. and
    makes sure the nested structure is the same, and also that all
    floating points match within the given tolerances.

    :params inp1: Structure 1 to compare.
    :params inp2: Structure 2 to compare.
    :params rel_tol: Maximum difference for being considered "close",
    relative to the magnitude of the input values as defined by math.isclose().
    :params abs_tol: Maximum difference for being considered "close",
    regardless of the magnitude of the input values as defined by
    math.isclose().

    :returns: bool indicating if structures don't match (False) or do match
    (True)
    """
    if isinstance(inp1, dict):
        if inp1.keys() != inp2.keys():
            return False
        for key in inp1.keys() & inp2.keys():
            val1 = inp1[key]
            val2 = inp2[key]
            if not compare_inputs(val1, val2,
                                  rel_tol=rel_tol, abs_tol=abs_tol):
                return False
    elif isinstance(inp1, float):
        if not isclose(inp1, inp2, rel_tol=rel_tol, abs_tol=abs_tol):
            return False
    elif not isinstance(inp1, str) and isinstance(inp1, Iterable):
        if len(inp1) != len(inp2):
            return False
        for val1, val2 in zip(inp1, inp2):
            if not compare_inputs(val1, val2,
                                rel_tol=rel_tol, abs_tol=abs_tol):
                return False
        else:
            if inp1 != inp2:
                return False

        return True


class Sigma:
    def __init__(self, iq, q_c, fxc, esknshape, nw, **inputs):
        """Inputs are used for cache invalidation, and are stored for each
           file.
        """
        self.iq = iq
        self.q_c = q_c
        self.fxc = fxc
        # We might as well allocate both from same array
        # in order to add and communicate to them faster.
        self._buf = np.zeros((2, *esknshape))
        # self-energies and derivatives:
        self.sigma_eskn, self.dsigma_eskn = self._buf

        eskwnshape = (*esknshape[:3], nw, esknshape[3])
        self.sigma_eskwn = np.zeros(eskwnshape, dtype=complex)
        self.inputs = inputs

    def sum(self, comm):
        comm.sum(self._buf)
        comm.sum(self.sigma_eskwn)

    def __iadd__(self, other):
        self.validate_inputs(other.inputs)
        self._buf += other._buf
        self.sigma_eskwn += other.sigma_eskwn
        return self

    def validate_inputs(self, inputs):
        equals = compare_inputs(inputs, self.inputs, rel_tol=1e-12,
                                abs_tol=1e-12)
        if not equals:
            raise RuntimeError('There exists a cache with mismatching input '
                               f'parameters: {inputs} != {self.inputs}.')

    @classmethod
    def fromdict(cls, dct):
        instance = cls(dct['iq'], dct['q_c'], dct['fxc'],
                       dct['sigma_eskn'].shape, dct['sigma_eskwn'].shape[3],
                       **dct['inputs'])
        instance.sigma_eskn[:] = dct['sigma_eskn']
        instance.dsigma_eskn[:] = dct['dsigma_eskn']
        instance.sigma_eskwn[:] = dct['sigma_eskwn']
        return instance

    def todict(self):
        return {'iq': self.iq,
                'q_c': self.q_c,
                'fxc': self.fxc,
                'sigma_eskn': self.sigma_eskn,
                'sigma_eskwn': self.sigma_eskwn,
                'dsigma_eskn': self.dsigma_eskn,
                'inputs': self.inputs}


class G0W0Outputs:
    def __init__(self, context, shape, ecut_e, sigma_eskn, dsigma_eskn,
                 sigma_eskwn, eps_skn, vxc_skn, exx_skn, f_skn):
        self.extrapolate(context, shape, ecut_e, sigma_eskn, dsigma_eskn)
        self.Z_skn = 1 / (1 - self.dsigma_skn)
        # G0W0 single-step.
        # If we want GW0 again, we need to grab the expressions
        # from e.g. e73917fca5b9dc06c899f00b26a7c46e7d6fa749
        # or earlier and use qp correctly.
        self.qp_skn = eps_skn + self.Z_skn * (
            -vxc_skn + exx_skn + self.sigma_skn)

        self.sigma_eskn = sigma_eskn
        self.dsigma_eskn = dsigma_eskn

        self.eps_skn = eps_skn
        self.vxc_skn = vxc_skn
        self.exx_skn = exx_skn
        self.f_skn = f_skn
        self.sigma_eskwn = sigma_eskwn

    def extrapolate(self, context, shape, ecut_e, sigma_eskn, dsigma_eskn):
        if len(ecut_e) == 1:
            self.sigma_skn = sigma_eskn[0]
            self.dsigma_skn = dsigma_eskn[0]
            self.sigr2_skn = None
            self.dsigr2_skn = None
            return

        from scipy.stats import linregress

        # Do linear fit of selfenergy vs. inverse of number of plane waves
        # to extrapolate to infinite number of plane waves

        context.print('', flush=False)
        context.print('Extrapolating selfenergy to infinite energy cutoff:',
                      flush=False)
        context.print('  Performing linear fit to %d points' % len(ecut_e))
        self.sigr2_skn = np.zeros(shape)
        self.dsigr2_skn = np.zeros(shape)
        self.sigma_skn = np.zeros(shape)
        self.dsigma_skn = np.zeros(shape)
        invN_i = ecut_e**(-3. / 2)
        for m in range(np.prod(shape)):
            s, k, n = np.unravel_index(m, shape)

            slope, intercept, r_value, p_value, std_err = \
                linregress(invN_i, sigma_eskn[:, s, k, n])

            self.sigr2_skn[s, k, n] = r_value**2
            self.sigma_skn[s, k, n] = intercept

            slope, intercept, r_value, p_value, std_err = \
                linregress(invN_i, dsigma_eskn[:, s, k, n])

            self.dsigr2_skn[s, k, n] = r_value**2
            self.dsigma_skn[s, k, n] = intercept

        if np.any(self.sigr2_skn < 0.9) or np.any(self.dsigr2_skn < 0.9):
            context.print('  Warning: Bad quality of linear fit for some ('
                          'n,k). ', flush=False)
            context.print('           Higher cutoff might be necessary.',
                          flush=False)

        context.print('  Minimum R^2 = %1.4f. (R^2 Should be close to 1)' %
                      min(np.min(self.sigr2_skn), np.min(self.dsigr2_skn)))

    def get_results_eV(self):
        results = {
            'f': self.f_skn,
            'eps': self.eps_skn * Ha,
            'vxc': self.vxc_skn * Ha,
            'exx': self.exx_skn * Ha,
```

```python
                'sigma': self.sigma_skn * Ha,
                'dsigma': self.dsigma_skn,
                'Z': self.Z_skn,
                'qp': self.qp_skn * Ha}

        results.update(
            sigma_eskn=self.sigma_eskn * Ha,
            dsigma_eskn=self.dsigma_eskn,
            sigma_eskwn=self.sigma_eskwn * Ha)

        if self.sigr2_skn is not None:
            assert self.dsigr2_skn is not None
            results['sigr2_skn'] = self.sigr2_skn
            results['dsigr2_skn'] = self.dsigr2_skn

        return results


class QSymmetryOp:
    def __init__(self, symno, U_cc, sign):
        self.symno = symno
        self.U_cc = U_cc
        self.sign = sign

    def apply(self, q_c):
        return self.sign * (self.U_cc @ q_c)

    def check_q_Q_symmetry(self, Q_c, q_c):
        d_c = self.apply(q_c) - Q_c
        assert np.allclose(d_c.round(), d_c)

    def get_M_vv(self, cell_cv):
        # We'll be inverting these cells a lot.
        # Should have an object with the cell and its inverse which does this.
        return cell_cv.T @ self.U_cc.T @ np.linalg.inv(cell_cv).T

    @classmethod
    def get_symops(cls, qd, iq, q_c):
        # Loop over all k-points in the BZ and find those that are
        # related to the current IBZ k-point by symmetry
        Q1 = qd.ibz2bz_k[iq]
        done = set()
        for Q2 in qd.bz2bz_ks[Q1]:
            if Q2 >= 0 and Q2 not in done:
                time_reversal = qd.time_reversal_k[Q2]
                symno = qd.sym_k[Q2]
                Q_c = qd.bzk_kc[Q2]

                symop = cls(
                    symno=symno,
                    U_cc=qd.symmetry.op_scc[symno],
                    sign=1 - 2 * time_reversal)

                symop.check_q_Q_symmetry(Q_c, q_c)
                # Q_c, symop = QSymmetryOp.from_qd(qd, Q2, q_c)
                yield Q_c, symop
                done.add(Q2)

    @classmethod
    def get_symop_from_kpair(cls, kd, qd, kpt1, kpt2):
        # from k-point pair kpt1, kpt2 get Q_c = kpt2-kpt1, corrsponding IBZ
        # k-point q_c, indexes iQ, iq and symmetry transformation relating
        # Q_c to q_c
        Q_c = kd.bzk_kc[kpt2.K] - kd.bzk_kc[kpt1.K]
        iQ = qd.where_is_q(Q_c, qd.bzk_kc)
        iq = qd.bz2ibz_k[iQ]
        q_c = qd.ibzk_kc[iq]

        # Find symmetry that transforms Q_c into q_c
        sym = qd.sym_k[iQ]
        U_cc = qd.symmetry.op_scc[sym]
        time_reversal = qd.time_reversal_k[iQ]
        sign = 1 - 2 * time_reversal
        symop = cls(sym, U_cc, sign)
        symop.check_q_Q_symmetry(Q_c, q_c)

        return symop, iq

    def apply_symop_q(self, qpd, pawcorr, kpt1, kpt2):
        # returns necessary quantities to get symmetry transformed
        # density matrix
        Q_G = phase_shifted_fft_indices(kpt1.k_c, kpt2.k_c, qpd,
                                        coordinate_transformation=self.apply)

        qG_Gv = qpd.get_reciprocal_vectors(add_q=True)
        M_vv = self.get_M_vv(qpd.gd.cell_cv)
        mypawcorr = pawcorr.remap_by_symop(self, qG_Gv, M_vv)

        return mypawcorr, Q_G


def get_nmG(kpt1, kpt2, mypawcorr, n, qpd, I_G, pair_calc, timer=None):
    if timer:
        timer.start('utcc and pawcorr multiply')
    ut1cc_R = kpt1.ut_nR[n].conj()
    C1_aGi = mypawcorr.multiply(kpt1.P_ani, band=n)
    if timer:
        timer.stop('utcc and pawcorr multiply')
    n_mG = pair_calc.calculate_pair_density(
        ut1cc_R, C1_aGi, kpt2, qpd, I_G)
    return n_mG


gw_logo = """\
  ___  _ _ _
 |   || | | |
 | | || | | |
 |__ ||_____|
 |___|
"""


def get_max_nblocks(world, calc, ecut):
    nblocks = world.size
    if not isinstance(calc, (str, Path)):
        raise Exception('Using a calulator is not implemented at '
                        'the moment, load from file!')
        # nblocks_calc = calc
    else:
        nblocks_calc = GPAW(calc)
    ngmax = []
    for q_c in nblocks_calc.wfs.kd.bzk_kc:
        qpd = SingleQPWDescriptor.from_q(q_c, np.min(ecut) / Ha,
                                         nblocks_calc.wfs.gd)
        ngmax.append(qpd.ngmax)
    nG = np.min(ngmax)

    while nblocks > nG**0.5 + 1 or world.size % nblocks != 0:
        nblocks -= 1

    mynG = (nG + nblocks - 1) // nblocks
    assert mynG * (nblocks - 1) < nG
    return nblocks


def get_frequencies(frequencies: dict | None,
                    domega0: float | None, omega2: float | None):
    if domega0 is not None or omega2 is not None:
        assert frequencies is None
        frequencies = {'type': 'nonlinear',
                       'domega0': 0.025 if domega0 is None else domega0,
                       'omega2': 10.0 if omega2 is None else omega2}
        warnings.warn(f'Please use frequencies={frequencies}')
    elif frequencies is None:
        frequencies = {'type': 'nonlinear',
                       'domega0': 0.025,
                       'omega2': 10.0}
    else:
        assert frequencies['type'] == 'nonlinear'
    return frequencies


def choose_ecut_things(ecut, ecut_extrapolation):
    if ecut_extrapolation is True:
        pct = 0.8
        necuts = 3
        ecut_e = ecut * (1 + (1. / pct - 1) * np.arange(necuts)[::-1] /
                         (necuts - 1))**(-2 / 3)
    elif isinstance(ecut_extrapolation, (list, np.ndarray)):
        ecut_e = np.array(np.sort(ecut_extrapolation))
        if not np.allclose(ecut, ecut_e[-1]):
            raise ValueError('ecut parameter must be the largest value'
                             'of ecut_extrapolation, when it is a list.')
        ecut = ecut_e[-1]
    else:
        ecut_e = np.array([ecut])
    return ecut, ecut_e


def select_kpts(kpts, kd):
    """Function to process input parameters that take a list of k-points given
    in different format and returns a list of indices of the corresponding
    k-points in the IBZ."""

    if kpts is None:
        # Do all k-points in the IBZ:
        return np.arange(kd.nibzkpts)

    if np.asarray(kpts).ndim == 1:
        return kpts

    # Find k-points:
    bzk_Kc = kd.bzk_kc
    indices = []
    for k_c in kpts:
        d_Kc = bzk_Kc - k_c
        d_Kc -= d_Kc.round()
        K = abs(d_Kc).sum(1).argmin()
        if not np.allclose(d_Kc[K], 0):
            raise ValueError('Could not find k-point: {k_c}'
                             .format(k_c=k_c))
        k = kd.bz2ibz_k[K]
        indices.append(k)
    return indices


class PairDistribution:
    def __init__(self, kptpair_factory, blockcomm, mysKn1n2):
        self.get_k_point = kptpair_factory.get_k_point
        self.kd = kptpair_factory.gs.kd
        self.blockcomm = blockcomm
        self.mysKn1n2 = mysKn1n2
        self.mykpts = [self.get_k_point(s, K, n1, n2)
                       for s, K, n1, n2 in self.mysKn1n2]

    def kpt_pairs_by_q(self, q_c, m1, m2):
        mykpts = self.mykpts
        for u, kpt1 in enumerate(mykpts):
            progress = u / len(mykpts)
            K2 = self.kd.find_k_plus_q(q_c, [kpt1.K])[0]
            kpt2 = self.get_k_point(kpt1.s, K2, m1, m2,
                                    blockcomm=self.blockcomm)

            yield progress, kpt1, kpt2


def distribute_k_points_and_bands(chi0_body_calc, band1, band2, kpts=None):
    """Distribute spins, k-points and bands.

    The attribute self.mysKn1n2 will be set to a list of (s, K, n1, n2)
    tuples that this process handles.
    """
    gs = chi0_body_calc.gs
    blockcomm = chi0_body_calc.blockcomm
    kncomm = chi0_body_calc.kncomm

    if kpts is None:
        kpts = np.arange(gs.kd.nbzkpts)

    # nbands is the number of bands for each spin/k-point combination.
    nbands = band2 - band1
    size = kncomm.size
    rank = kncomm.rank
    ns = gs.nspins
    nk = len(kpts)
    n = (ns * nk * nbands + size - 1) // size
    i1 = min(rank * n, ns * nk * nbands)
    i2 = min(i1 + n, ns * nk * nbands)

    mysKn1n2 = []
    i = 0
    for s in range(ns):
        for K in kpts:
            n1 = min(max(0, i1 - i), nbands)
            n2 = min(max(0, i2 - i), nbands)
            if n1 != n2:
                mysKn1n2.append((s, K, n1 + band1, n2 + band1))
            i += nbands

    p = chi0_body_calc.context.print
    p('BZ k-points:', gs.kd, flush=False)
    p('Distributing spins, k-points and bands (%d x %d x %d)' %
      (ns, nk, nbands), 'over %d process%s' %
      (kncomm.size, ['es', ''][kncomm.size == 1]),
      flush=False)
    p('Number of blocks:', blockcomm.size)

    return PairDistribution(
        chi0_body_calc.kptpair_factory, blockcomm, mysKn1n2)


class G0W0Calculator:
    def __init__(self, filename='gw', *,
                 wd,
                 chi0calc,
                 wcalc,
                 kpts, bands, nbands=None,
                 fxc_modes,
                 eta,
                 ecut_e,
                 frequencies=None,
                 exx_vxc_calculator,
                 qcache,
                 ppa=False,
                 mpa=None,
                 evaluate_sigma=None):
```

```python
"""G0W0 calculator, initialized through G0W0 object.

The G0W0 calculator is used to calculate the quasi
particle energies through the G0W0 approximation for a number
of states.

Parameters
----------
filename: str
    Base filename of output files.
wcalc: WCalculator object
    Defines the calculator for computing the screened interaction
kpts: list
    List of indices of the IBZ k-points to calculate the quasi particle
    energies for.
bands:
    Range of band indices, like (n1, n2), to calculate the quasi
    particle energies for. Bands n where n1<=n<n2 will be
    calculated.  Note that the second band index is not included.
frequencies:
    Input parameters for frequency_grid.
    Can be array of frequencies to evaluate the response function at
    or dictionary of parameters for build-in nonlinear grid
    (see :ref:`frequency grid`).
ecut_e: array(float)
    Plane wave cut-off energies in eV. Defined with choose_ecut_things
nbands: int
    Number of bands to use in the calculation. If None, the number will
    be determined from :ecut: to yield a number close to the number of
    plane waves used.
do_GW_too: bool
    When carrying out a calculation including vertex corrections, it
    is possible to get the standard GW results at the same time
    (almost for free).
ppa: bool
    Use Godby-Needs plasmon-pole approximation for screened interaction
    and self-energy (reformulated as mpa with npoles = 1)
mpa: dict
    Use multipole approximation for screened interaction
    and self-energy [PRB 104, 115157 (2021)]
    This method uses a sampling along one or two lines in the complex
    frequency plane.

    MPA parameters
    ----------
    npoles: Number of poles (positive integer generally lower than 15)
    parallel_lines: How many (1-2) parallel lines to the real frequency
                    axis the sampling has.
    wrange: Real interval defining the range of energy along the real
            frequency axis.
    alpha: exponent of the power distribution of points along the real
           frequency axis [PRB 107, 155130 (2023)]
    varpi: Distance of the second line to the real axis.
    eta0:  Imaginary part of the first point of the first line.
    eta_rest: Imaginary part of the rest of the points of the first
              line.
evaluate_sigma: array(float)
    List of frequencies (in eV), where to evaluate the frequency
    dependent self energy for each k-point and band involved in the
    sigma-evaluation. This will be done in addition to evaluating the
    normal self-energy quasiparticle matrix elements in G0W0
    approximation.
"""
self.chi0calc = chi0calc
self.wcalc = wcalc
self.context = self.wcalc.context
self.ppa = ppa
self.mpa = mpa
if evaluate_sigma is None:
    evaluate_sigma = np.array([])
self.evaluate_sigma = evaluate_sigma
self.qcache = qcache

# Note: self.wd should be our only representation of the frequencies.
# We should therefore get rid of self.frequencies.
# It is currently only used by the restart code,
# so should be easy to remove after some further adaptation.
self.wd = wd
self.frequencies = frequencies

self.ecut_e = ecut_e / Ha

self.context.print(gw_logo)

if self.chi0calc.gs.metallic:
    self.context.print('WARNING: \n'
                       'The current GW implementation cannot'
                       ' handle intraband screening. \n'
                       'This results in poor k-point'
                       ' convergence for metals')

self.fxc_modes = fxc_modes

if self.fxc_modes[0] != 'GW':
    assert self.wcalc.xckernel.xc != 'RPA'

if len(self.fxc_modes) == 2:
    # With multiple fxc_modes, we previously could do only
    # GW plus one other fxc_mode.  Now we can have any set
    # of modes, but whether things are consistent or not may
    # depend on how wcalc is configured.
    assert 'GW' in self.fxc_modes
    assert self.wcalc.xckernel.xc != 'RPA'

self.filename = filename
self.eta = eta / Ha

self.kpts = kpts
self.bands = bands

b1, b2 = self.bands
self.shape = (self.wcalc.gs.nspins, len(self.kpts), b2 - b1)

self.nbands = nbands

if self.wcalc.gs.nspins != 1:
    for fxc_mode in self.fxc_modes:
        if fxc_mode != 'GW':
            raise RuntimeError('Including a xc kernel does not '
                               'currently work for spin-polarized '
                               f'systems. Invalid fxc_mode {fxc_mode}.'
                               )

self.pair_distribution = distribute_k_points_and_bands(
    self.chi0calc.chi0_body_calc, b1, b2,
    self.chi0calc.gs.kd.ibz2bz_k[self.kpts])

self.print_parameters(kpts, b1, b2)

self.exx_vxc_calculator = exx_vxc_calculator

p = self.context.print
if self.ppa:
    p('Using Godby-Needs plasmon-pole approximation:')
    p('  Fitting energy: i*E0, E0 = '
      f'{self.wd.omega_w[1].imag:.3f} Hartree')
elif self.mpa:
    omega_w = self.chi0calc.wd.omega_w
```

```
            p('Using multipole approximation:')
            p(f'  Number of poles: {len(omega_w) // 2}')
            p(f'  Energy range: Re(E[-1]) = {omega_w[-1].real:.3f} Hartree')
            p('  Imaginary range: Im(E[-1]) = '
              f'{self.wd.omega_w[-1].imag:.3f} Hartree')
            p('  Imaginary shift: Im(E[1]) = '
              f'{self.wd.omega_w[1].imag:.3f} Hartree')
            p('  Imaginary Origin shift: Im(E[0])'
              f'= {self.wd.omega_w[0].imag:.3f} Hartree')
        else:
            self.context.print('Using full-frequency real axis integration')

    def print_parameters(self, kpts, b1, b2):
        isl = ['',
               'Quasi particle states:']
        if kpts is None:
            isl.append('All k-points in IBZ')
        else:
            kptstxt = ', '.join([f'{k:d}' for k in self.kpts])
            isl.append(f'k-points (IBZ indices): [{kptstxt}]')
        isl.extend([f'Band range: ({b1:d}, {b2:d})',
                    '',
                    'Computational parameters:'])
        if len(self.ecut_e) == 1:
            isl.append(
                'Plane wave cut-off: '
                f'{self.chi0calc.chi0_body_calc.ecut * Ha:g} eV')
        else:
            assert len(self.ecut_e) > 1
            isl.append('Extrapolating to infinite plane wave cut-off using '
                       'points at:')
            for ec in self.ecut_e:
                isl.append(f'  {ec * Ha:.3f} eV')
        isl.extend([f'Number of bands: {self.nbands:d}',
                    f'Coulomb cutoff: {self.wcalc.coulomb.truncation}',
                    f'Broadening: {self.eta * Ha:g} eV',
                    '',
                    f'fxc modes: {", ".join(sorted(self.fxc_modes))}',
                    f'Kernel: {self.wcalc.xckernel.xc}'])
        self.context.print('\n'.join(isl))

    def get_eps_and_occs(self):
        eps_skn = np.empty(self.shape)  # KS-eigenvalues
        f_skn = np.empty(self.shape)  # occupation numbers

        nspins = self.wcalc.gs.nspins
        b1, b2 = self.bands
        for i, k in enumerate(self.kpts):
            for s in range(nspins):
                u = s + k * nspins
                kpt = self.wcalc.gs.kpt_u[u]
                eps_skn[s, i] = kpt.eps_n[b1:b2]
                f_skn[s, i] = kpt.f_n[b1:b2] / kpt.weight

        return eps_skn, f_skn

    @timer('G0W0')
    def calculate(self, qpoints=None):
        """Starts the G0W0 calculation.

        qpoints: list[int]
            Set of q-points to calculate.

        Returns a dict with the results with the following key/value pairs:

        ===========  =============================================
        key          value
        ===========  =============================================
        ``f``        Occupation numbers
        ``eps``      Kohn-Sham eigenvalues in eV
        ``vxc``      Exchange-correlation
                     contributions in eV
        ``exx``      Exact exchange contributions in eV
        ``sigma``    Self-energy contributions in eV
        ``dsigma``   Self-energy derivatives
        ``sigma_e``  Self-energy contributions in eV
                     used for ecut extrapolation
        ``Z``        Renormalization factors
        ``qp``       Quasi particle (QP) energies in eV
        ``iqp``      GW0/GW: QP energies for each iteration in eV
        ===========  =======================================

        All the values are ``ndarray``'s of shape
        (spins, IBZ k-points, bands)."""

        qpoints = set(qpoints) if qpoints else None

        if qpoints is None:
            self.context.print('Summing all q:')
        else:
            qpt_str = ' '.join(map(str, qpoints))
            self.context.print(f'Calculating following q-points: {qpt_str}')
        self.calculate_q_points(qpoints=qpoints)
        if qpoints is not None:
            return f'A partial result of q-points: {qpt_str}'
        sigmas = self.read_sigmas()
        self.all_results = self.postprocess(sigmas)
        # Note: self.results is a pointer pointing to one of the results,
        # for historical reasons.

        self.savepckl()
        return self.results

    def postprocess(self, sigmas):
        all_results = {}
        for fxc_mode, sigma in sigmas.items():
            all_results[fxc_mode] = self.postprocess_single(fxc_mode, sigma)

        self.print_results(all_results)
        return all_results

    def read_sigmas(self):
        if self.context.comm.rank == 0:
            sigmas = self._read_sigmas()
        else:
            sigmas = None

        return broadcast(sigmas, comm=self.context.comm)

    def _read_sigmas(self):
        assert self.context.comm.rank == 0

        # Integrate over all q-points, and accumulate the quasiparticle shifts
        for iq, q_c in enumerate(self.wcalc.qd.ibzk_kc):
            key = str(iq)

            sigmas_contrib = self.get_sigmas_dict(key)

            if iq == 0:
                sigmas = sigmas_contrib
            else:
                for fxc_mode in self.fxc_modes:
                    sigmas[fxc_mode] += sigmas_contrib[fxc_mode]

        return sigmas

    def get_sigmas_dict(self, key):
        assert self.context.comm.rank == 0
```

```python
        return {fxc_mode: Sigma.fromdict(sigma)
                for fxc_mode, sigma in self.qcache[key].items()}

    def postprocess_single(self, fxc_name, sigma):
        output = self.calculate_g0w0_outputs(sigma)
        return output.get_results_eV()

    def savepckl(self):
        """Save outputs to pckl files and return paths to those files."""
        # Note: this is always called, but the paths aren't returned
        # to the caller.  Calling it again then overwrites the files.
        #
        # TODO:
        #  * Replace with JSON
        #  * Save to different files or same file?
        #  * Move this functionality to g0w0 result object
        paths = {}
        for fxc_mode in self.fxc_modes:
            path = Path(f'{self.filename}_results_{fxc_mode}.pckl')
            with paropen(path, 'wb', comm=self.context.comm) as fd:
                pickle.dump(self.all_results[fxc_mode], fd, 2)
            paths[fxc_mode] = path

        # Do not return paths to caller before we know they all exist:
        self.context.comm.barrier()
        return paths

    @property
    def nqpts(self):
        """Returns the number of q-points in the system."""
        return len(self.wcalc.qd.ibzk_kc)

    @timer('evaluate sigma')
    def calculate_q(self, ie, k, kpt1, kpt2, qpd, Wdict,
                    *, symop, sigmas, blocks1d, pawcorr):
        """Calculates the contribution to the self-energy and its derivative
        for a given set of k-points, kpt1 and kpt2."""
        mypawcorr, I_G = symop.apply_symop_q(qpd, pawcorr, kpt1, kpt2)
        if debug:
            N_c = qpd.gd.N_c
            i_cG = symop.apply(np.unravel_index(qpd.Q_qG[0], N_c))
            bzk_kc = self.wcalc.gs.kd.bzk_kc
            Q_c = bzk_kc[kpt2.K] - bzk_kc[kpt1.K]
            shift0_c = Q_c - symop.apply(qpd.q_c)
            self.check(ie, i_cG, shift0_c, N_c, Q_c, mypawcorr)

        for n in range(kpt1.n2 - kpt1.n1):
            eps1 = kpt1.eps_n[n]
            self.context.timer.start('get_nmG')
            n_mG = get_nmG(kpt1, kpt2, mypawcorr,
                           n, qpd, I_G, self.chi0calc.pair_calc)
            self.context.timer.stop('get_nmG')

            if symop.sign == 1:
                n_mG = n_mG.conj()

            f_m = kpt2.f_n
            deps_m = eps1 - kpt2.eps_n

            nn = kpt1.n1 + n - self.bands[0]

            assert set(Wdict) == set(sigmas)

            for fxc_mode in self.fxc_modes:
                sigma = sigmas[fxc_mode]
                Wmodel = Wdict[fxc_mode]

                # m is band index of all (both unoccupied and occupied) wave
                # functions in G
                for m, (deps, f, n_G) in enumerate(zip(deps_m, f_m, n_mG)):
                    # 2 * f - 1 will be used to select the branch of Hilbert
                    # transform, see get_HW of screened_interaction.py
                    # at FullFrequencyHWModel class.

                    nc_G = n_G.conj()
                    myn_G = n_G[blocks1d.myslice]

                    if self.evaluate_sigma is not None:
                        for w, omega in enumerate(self.evaluate_sigma):
                            S_GG, _ = Wmodel.get_HW(deps - eps1 + omega, f)
                            if S_GG is None:
                                continue
                            # print(myn_G.shape, S_GG.shape, nc_G.shape)
                            sigma.sigma_eskwn[ie, kpt1.s, k, w, nn] += \
                                myn_G @ S_GG @ nc_G

                    self.context.timer.start('Wmodel.get_HW')
                    S_GG, dSdw_GG = Wmodel.get_HW(deps, f)
                    self.context.timer.stop('Wmodel.get_HW')
                    if S_GG is None:
                        continue

                    # ie: ecut index for extrapolation
                    # kpt1.s: spin index of *
                    # k: k-point index of *
                    # nn: band index of *
                    # * wave function, where the sigma expectation value is
                    # evaluated
                    slot = ie, kpt1.s, k, nn
                    self.context.timer.start('n_G @ S_GG @ n_G')
                    sigma.sigma_eskn[slot] += (myn_G @ S_GG @ nc_G).real
                    sigma.dsigma_eskn[slot] += (myn_G @ dSdw_GG @ nc_G).real
                    self.context.timer.stop('n_G @ S_GG @ n_G')

    def check(self, ie, i_cG, shift0_c, N_c, Q_c, pawcorr):
        # Can we delete this check? XXX
        assert np.allclose(shift0_c.round(), shift0_c)
        shift0_c = shift0_c.round().astype(int)
        I0_G = np.ravel_multi_index(i_cG - shift0_c[:, None], N_c, 'wrap')
        qpd = SingleQPWDescriptor.from_q(Q_c, self.ecut_e[ie],
                                         self.wcalc.gs.gd)
        G_I = np.empty(N_c.prod(), int)
        G_I[:] = -1
        I1_G = qpd.Q_qG[0]
        G_I[I1_G] = np.arange(len(I0_G))
        G_G = G_I[I0_G]
        # This indexing magic should definitely be moved to a method.
        # What on earth is it really?

        assert len(I0_G) == len(I1_G)
        assert (G_G >= 0).all()
        pairden_paw_corr = self.wcalc.gs.pair_density_paw_corrections
        pawcorr_wcalc1 = pairden_paw_corr(qpd)
        assert pawcorr.almost_equal(pawcorr_wcalc1, G_G)

    def calculate_q_points(self, qpoints):
        """Main loop over irreducible Brillouin zone points.
        Handles restarts of individual qpoints using FileCache from ASE,
        and subsequently calls calculate_q."""

        pb = ProgressBar(self.context.fd)

        self.context.timer.start('W')
        self.context.print('\nCalculating screened Coulomb potential')
        self.context.print(self.wcalc.coulomb.description())

        chi0calc = self.chi0calc
        self.context.print(self.wd)
        # Find maximum size of chi-0 matrices:
        nGmax = max(count_reciprocal_vectors(chi0calc.chi0_body_calc.ecut,
                                             self.wcalc.gs.gd, q_c)
                    for q_c in self.wcalc.qd.ibzk_kc)
        nw = len(self.wd)

        size = self.chi0calc.chi0_body_calc.integrator.blockcomm.size

        mynGmax = (nGmax + size - 1) // size
        mynw = (nw + size - 1) // size

        # some memory sizes...
        if self.context.comm.rank == 0:
            siz = (nw * mynGmax * nGmax +
                   max(mynw * nGmax, nw * mynGmax) * nGmax) * 16
            sizA = (nw * nGmax * nGmax + nw * nGmax * nGmax) * 16
            self.context.print(
                '  memory estimate for chi0: local=%.2f MB, global=%.2f MB'
                % (siz / 1024**2, sizA / 1024**2))

        if self.context.comm.rank == 0 and qpoints is None:
            self.context.print('Removing empty qpoint cache files...')
            self.qcache.strip_empties()

        self.context.comm.barrier()

        # Need to pause the timer in between iterations
        self.context.timer.stop('W')

        with broadcast_exception(self.context.comm):
            if self.context.comm.rank == 0:
                for key, sigmas in self.qcache.items():
                    if qpoints and int(key) not in qpoints:
                        continue
                    sigmas = {fxc_mode: Sigma.fromdict(sigma)
                              for fxc_mode, sigma in sigmas.items()}
                    for fxc_mode, sigma in sigmas.items():
                        sigma.validate_inputs(self.get_validation_inputs())

        for iq, q_c in enumerate(self.wcalc.qd.ibzk_kc):
            # If a list of q-points is specified,
            # skip the q-points not in the list
            if qpoints and (iq not in qpoints):
                continue
            with ExitStack() as stack:
                if self.context.comm.rank == 0:
                    qhandle = stack.enter_context(self.qcache.lock(str(iq)))
                    skip = qhandle is None
                else:
                    skip = False

                skip = broadcast(skip, comm=self.context.comm)

                if skip:
                    continue

                result = self.calculate_q_point(iq, q_c, pb, chi0calc)

                if self.context.comm.rank == 0:
                    qhandle.save(result)
        pb.finish()

    def calculate_q_point(self, iq, q_c, pb, chi0calc):
        # Reset calculation
        sigmashape = (len(self.ecut_e), *self.shape)
        sigmas = {fxc_mode: Sigma(iq, q_c, fxc_mode, sigmashape,
                  len(self.evaluate_sigma),
                  **self.get_validation_inputs())
                  for fxc_mode in self.fxc_modes}

        chi0 = chi0calc.create_chi0(q_c)

        m1 = chi0calc.gs.nocc1
        for ie, ecut in enumerate(self.ecut_e):
            self.context.timer.start('W')

            # First time calculation
            if ecut == chi0.qpd.ecut:
                # Nothing to cut away:
                m2 = self.nbands
            else:
                m2 = int(self.wcalc.gs.volume * ecut**1.5
                         * 2**0.5 / 3 / pi**2)
                if m2 > self.nbands:
                    raise ValueError(f'Trying to extrapolate ecut to'
                                     f'larger number of bands ({m2})'
                                     f' than there are bands '
                                     f'({self.nbands}).')
            qpdi, Wdict, blocks1d, pawcorr = self.calculate_w(
                chi0calc, q_c, chi0,
                m1, m2, ecut, iq)
            m1 = m2

            self.context.timer.stop('W')

            for nQ, (bzq_c, symop) in enumerate(QSymmetryOp.get_symops(
                    self.wcalc.qd, iq, q_c)):

                for (progress, kpt1, kpt2)\
                        in self.pair_distribution.kpt_pairs_by_q(bzq_c, 0, m2):
                    pb.update((nQ + progress) / self.wcalc.qd.mynk)

                    k1 = self.wcalc.gs.kd.bz2ibz_k[kpt1.K]
                    i = self.kpts.index(k1)
                    self.calculate_q(ie, i, kpt1, kpt2, qpdi, Wdict,
                                     symop=symop,
                                     sigmas=sigmas,
                                     blocks1d=blocks1d,
                                     pawcorr=pawcorr)

        for sigma in sigmas.values():
            sigma.sum(self.context.comm)

        return sigmas

    def get_validation_inputs(self):
        return {'kpts': self.kpts,
                'bands': list(self.bands),
                'nbands': self.nbands,
                'ecut_e': list(self.ecut_e),
                'frequencies': self.frequencies,
                'fxc_modes': self.fxc_modes,
                'integrate_gamma': repr(self.wcalc.integrate_gamma)}

    @timer('calculate_w')
    def calculate_w(self, chi0calc, q_c, chi0,
                    m1, m2, ecut,
                    iq):
        """Calculates the screened potential for a specified q-point."""

        chi0calc.chi0_body_calc.print_info(chi0.qpd)
        chi0calc.update_chi0(chi0, m1, m2, range(self.wcalc.gs.nspins))

        Wdict = {}

        for fxc_mode in self.fxc_modes:
            rqpd = chi0.qpd.copy_with(ecut=ecut)  # reduced qpd
            rchi0 = chi0.copy_with_reduced_pd(rqpd)
            Wdict[fxc_mode] = self.wcalc.get_HW_model(rchi0,
                                                      fxc_mode=fxc_mode)
            if (chi0calc.chi0_body_calc.pawcorr is not None and
                    rqpd.ecut < chi0.qpd.ecut):
                pw_map = PWMapping(rqpd, chi0.qpd)

                """This is extremely bad behaviour! G0W0Calculator
                   should not change properties on the
                   Chi0BodyCalculator! Change in the future! XXX"""
                chi0calc.chi0_body_calc.pawcorr = \
                    chi0calc.chi0_body_calc.pawcorr.reduce_ecut(pw_map.G2_G1)

        # Create a blocks1d for the reduced plane-wave description
        blocks1d = Blocks1D(chi0.body.blockdist.blockcomm, rqpd.ngmax)

        return rqpd, Wdict, blocks1d, chi0calc.chi0_body_calc.pawcorr

    @timer('calculate_vxc_and_exx')
    def calculate_vxc_and_exx(self):
        return self.exx_vxc_calculator.calculate(
            n1=self.bands[0], n2=self.bands[1],
            kpt_indices=self.kpts)

    def print_results(self, results):
        description = ['f:      Occupation numbers',
                       'eps:     KS-eigenvalues [eV]',
                       'vxc:     KS vxc [eV]',
                       'exx:     Exact exchange [eV]',
                       'sigma:   Self-energies [eV]',
                       'dsigma:  Self-energy derivatives',
                       'Z:       Renormalization factors',
                       'qp:      QP-energies [eV]']

        self.context.print('\nResults:')
        for line in description:
            self.context.print(line)

        b1, b2 = self.bands
        names = [line.split(':', 1)[0] for line in description]
        ibzk_kc = self.wcalc.gs.kd.ibzk_kc
        for s in range(self.wcalc.gs.nspins):
            for i, ik in enumerate(self.kpts):
                self.context.print(
                    '\nk-point ' + '{} ({}): ({:.3f}, {:.3f}, '
                    '{:.3f})'.format(i, ik, *ibzk_kc[ik]) +
                    '                ' + self.fxc_modes[0])
                self.context.print('band' + ''.join(f'{name:>8}'
                                                    for name in names))

                def actually_print_results(resultset):
                    for n in range(b2 - b1):
                        self.context.print(
                            f'{n + b1:4}' +
                            ''.join('{:8.3f}'.format(
                                resultset[name][s, i, n]) for name in names))

                for fxc_mode in results:
                    self.context.print(fxc_mode.rjust(69))
                    actually_print_results(results[fxc_mode])

        self.context.write_timer()

    def calculate_g0w0_outputs(self, sigma):
        eps_skn, f_skn = self.get_eps_and_occs()
        vxc_skn, exx_skn = self.calculate_vxc_and_exx()
        kwargs = dict(
            context=self.context,
            shape=self.shape,
```

```python
            ecut_e=self.ecut_e,
            eps_skn=eps_skn,
            vxc_skn=vxc_skn,
            exx_skn=exx_skn,
            f_skn=f_skn)

        return G0W0Outputs(sigma_eskn=sigma.sigma_eskn,
                           dsigma_eskn=sigma.dsigma_eskn,
                           sigma_eskwn=sigma.sigma_eskwn,
                           **kwargs)


def choose_bands(bands, relbands, nvalence, nocc):
    if bands is not None and relbands is not None:
        raise ValueError('Use bands or relbands!')

    if relbands is not None:
        bands = [nvalence // 2 + b for b in relbands]

    if bands is None:
        bands = [0, nocc]

    return bands


class G0W0(G0W0Calculator):
    def __init__(self, calc, filename='gw',
                 ecut=150.0,
                 ecut_extrapolation=False,
                 xc='RPA',
                 ppa=False,
                 mpa=None,
                 E0=Ha,
                 eta=0.1,
                 nbands=None,
                 bands=None,
                 relbands=None,
                 frequencies=None,
                 domega0=None,  # deprecated
                 omega2=None,  # deprecated
                 nblocks=1,
                 nblocksmax=False,
                 kpts=None,
                 world=mpi.world,
                 timer=None,
                 fxc_mode='GW',
                 fxc_modes=None,
                 truncation=None,
                 integrate_gamma='sphere',
                 q0_correction=False,
                 do_GW_too=False,
                 output_prefix=None,
                 **kwargs):
        """G0W0 calculator wrapper.

        The G0W0 calculator is used to calculate the quasi
        particle energies through the G0W0 approximation for a number
        of states.

        Parameters
        ----------
        calc:
            Filename of saved calculator object.
        filename: str
            Base filename (a prefix) of output files.
        kpts: list
            List of indices of the IBZ k-points to calculate the quasi particle
            energies for.
        bands:
```

```
    Range of band indices, like (n1, n2), to calculate the quasi
    particle energies for. Bands n where n1<=n<n2 will be
    calculated.  Note that the second band index is not included.
relbands:
    Same as *bands* except that the numbers are relative to the
    number of occupied bands.
    E.g. (-1, 1) will use HOMO+LUMO.
frequencies:
    Input parameters for the nonlinear frequency descriptor.
ecut: float
    Plane wave cut-off energy in eV.
ecut_extrapolation: bool or list
    If set to True an automatic extrapolation of the selfenergy to
    infinite cutoff will be performed based on three points
    for the cutoff energy.
    If an array is given, the extrapolation will be performed based on
    the cutoff energies given in the array.
nbands: int
    Number of bands to use in the calculation. If None, the number will
    be determined from :ecut: to yield a number close to the number of
    plane waves used.
ppa: bool
    Sets whether the Godby-Needs plasmon-pole approximation for the
    dielectric function should be used.
 mpa: dict
    Sets whether the multipole approximation for the response
    function should be used.
xc: str
    Kernel to use when including vertex corrections.
fxc_mode: str
    Where to include the vertex corrections; polarizability and/or
    self-energy. 'GWP': Polarizability only, 'GWS': Self-energy only,
    'GWG': Both.
do_GW_too: bool
    When carrying out a calculation including vertex corrections, it
    is possible to get the standard GW results at the same time
    (almost for free).
truncation: str
    Coulomb truncation scheme. Can be either 2D, 1D, or 0D.
integrate_gamma: str or dict
    Method to integrate the Coulomb interaction.

    The default is 'sphere'. If 'reduced' key is not given,
    it defaults to False.

    {'type': 'sphere'} or 'sphere':
        Analytical integration of q=0, G=0 1/q^2 integrand in a sphere
        matching the volume of a single q-point.
        Used to be integrate_gamma=0.

    {'type': 'reciprocal'} or 'reciprocal':
        Numerical integration of q=0, G=0 1/q^2 integral in a volume
        resembling the reciprocal cell (parallelpiped).
        Used to be integrate_gamma=1.

    {'type': 'reciprocal', 'reduced':True} or 'reciprocal2D':
        Numerical integration of q=0, G=0 1/q^2 integral in a area
        resembling the reciprocal 2D cell (parallelogram) to be used
        to be usedwith 2D systems.
        Used to be integrate_gamma=2.

    {'type': '1BZ'} or '1BZ':
        Numerical integration of q=0, G=0 1/q^2 integral in a volume
        resembling the Wigner-Seitz cell of the reciprocal lattice
        (voronoi). More accurate than 'reciprocal'.

        A. Guandalini, P. D’Amico, A. Ferretti and D. Varsano:
        npj Computational Materials volume 9, Article number: 44 (2023)
    {'type': '1BZ', 'reduced': True} or '1BZ2D':
        Same as above, but everything is done in 2D (for 2D systems).

    {'type': 'WS'} or 'WS':
        The most accurate method to use for bulk systems.
        Instead of numerically integrating only q=0, G=0, all (q,G)-
        pairs participate to the truncation, which is done in real
        space utilizing the Wigner-Seitz truncation in the
        Born-von-Karmann supercell of the system.

        Numerical integration of q=0, G=0 1/q^2 integral in a volume
        resembling the Wigner-Seitz cell of the reciprocal lattice
        (Voronoi). More accurate than 'reciprocal'.

        R. Sundararaman and T. A. Arias: Phys. Rev. B 87, 165122 (2013)
E0: float
    Energy (in eV) used for fitting in the plasmon-pole approximation.
q0_correction: bool
    Analytic correction to the q=0 contribution applicable to 2D
    systems.
nblocks: int
    Number of blocks chi0 should be distributed in so each core
    does not have to store the entire matrix. This is to reduce
    memory requirement. nblocks must be less than or equal to the
    number of processors.
nblocksmax: bool
    Cuts chi0 into as many blocks as possible to reduce memory
    requirements as much as possible.
output_prefix: None | str
    Where to direct the txt output. If set to None (default),
    will be deduced from filename (the default output prefix).
    This is to allow multiple processes to work on same cache
    (given by filename-prefix), while writing to different out
    files.
"""
if fxc_mode:
    assert fxc_modes is None
if fxc_modes:
    assert fxc_mode is None

frequencies = get_frequencies(frequencies, domega0, omega2)

integrate_gamma = GammaIntegrationMode(integrate_gamma)

# We pass a serial communicator because the parallel handling
# is somewhat wonky, we'd rather do that ourselves:
try:
    qcache = FileCache(f'qcache_{filename}',
                       comm=mpi.SerialCommunicator())
except TypeError as err:
    raise RuntimeError(
        'File cache requires ASE master '
        'from September 20 2022 or newer.  '
        'You may need to pull newest ASE.') from err

mode = 'a' if qcache.filecount() > 1 else 'w'

# (calc can not actually be a calculator at all.)
gpwfile = Path(calc)

output_prefix = filename or output_prefix
context = ResponseContext(txt=output_prefix + '.txt',
                          comm=world, timer=timer)
gs = ResponseGroundStateAdapter.from_gpw_file(gpwfile)

# Check if nblocks is compatible, adjust if not
if nblocksmax:
    nblocks = get_max_nblocks(context.comm, gpwfile, ecut)
```

```python
kpts = list(select_kpts(kpts, gs.kd))

ecut, ecut_e = choose_ecut_things(ecut, ecut_extrapolation)

if nbands is None:
    nbands = int(gs.volume * (ecut / Ha)**1.5 * 2**0.5 / 3 / pi**2)
else:
    if ecut_extrapolation:
        raise RuntimeError(
            'nbands cannot be supplied with ecut-extrapolation.')

if ppa:
    # ppa reformulated as mpa with one pole
    mpa = {'npoles': 1, 'wrange': [0, 0], 'varpi': E0,
           'eta0': 1e-6, 'eta_rest': Ha, 'alpha': 1}

if mpa:

    frequencies = mpa_frequency_sampling(**mpa)

    parameters = {'eta': 1e-6,
                  'hilbert': False,
                  'timeordered': False}

else:
    # use nonlinear frequency grid
    frequencies = get_frequencies(frequencies, domega0, omega2)

    parameters = {'eta': eta,
                  'hilbert': True,
                  'timeordered': True}
wd = get_frequency_descriptor(frequencies, gs=gs, nbands=nbands)

wcontext = context.with_txt(output_prefix + '.w.txt', mode=mode)

chi0calc = Chi0Calculator(
    gs, wcontext, nblocks=nblocks,
    wd=wd,
    nbands=nbands,
    ecut=ecut,
    intraband=False,
    **parameters)

bands = choose_bands(bands, relbands, gs.nvalence, chi0calc.gs.nocc2)

coulomb = CoulombKernel.from_gs(gs, truncation=truncation)
# XXX eta needs to be converted to Hartree here,
# XXX and it is also converted to Hartree at superclass constructor
# XXX called below. This needs to be cleaned up.
wcalc = initialize_w_calculator(chi0calc, wcontext,
                                mpa=mpa,
                                xc=xc,
                                E0=E0, eta=eta / Ha, coulomb=coulomb,
                                integrate_gamma=integrate_gamma,
                                q0_correction=q0_correction)

if fxc_mode:
    fxc_modes = [fxc_mode]

if do_GW_too:
    fxc_modes.append('GW')

exx_vxc_calculator = EXXVXCCalculator(
    gpwfile,
    snapshotfile_prefix=filename)

super().__init__(filename=filename,
                 wd=wd,
                 chi0calc=chi0calc,
                                  wcalc=wcalc,
                                  ecut_e=ecut_e,
                                  eta=eta,
                                  fxc_modes=fxc_modes,
                                  nbands=nbands,
                                  bands=bands,
                                  frequencies=frequencies,
                                  kpts=kpts,
                                  exx_vxc_calculator=exx_vxc_calculator,
                                  qcache=qcache,
                                  ppa=ppa,
                                  mpa=mpa,
                                  **kwargs)

    @property
    def results_GW(self):
        # Compatibility with old "do_GW_too" behaviour
        if 'GW' in self.fxc_modes and self.fxc_modes[0] != 'GW':
            return self.all_results['GW']

    @property
    def results(self):
        return self.all_results[self.fxc_modes[0]]


class EXXVXCCalculator:
    """EXX and Kohn-Sham XC contribution."""

    def __init__(self, gpwfile, snapshotfile_prefix):
        self._gpwfile = gpwfile
        self._snapshotfile_prefix = snapshotfile_prefix

    def calculate(self, n1, n2, kpt_indices):
        _, vxc_skn, exx_skn = non_self_consistent_eigenvalues(
            self._gpwfile,
            'EXX',
            n1, n2,
            kpt_indices=kpt_indices,
            snapshot=f'{self._snapshotfile_prefix}-vxc-exx.json',
        )
        return vxc_skn / Ha, exx_skn / Ha
```