```
"""This module defines Coulomb and XC kernels for the response model.
"""
import warnings
import numpy as np
from ase.dft import monkhorst_pack
from gpaw.response.pair_functions import SingleQPWDescriptor
from gpaw.response.dyson import PWKernel
from gpaw.kpt_descriptor import to1bz


class NewCoulombKernel(PWKernel):
    def __init__(self, Vbare_G):
        self.Vbare_G = Vbare_G

    @classmethod
    def from_qpd(cls, qpd, **kwargs):
        return cls(get_coulomb_kernel(qpd, **kwargs))

    def get_number_of_plane_waves(self):
        return len(self.Vbare_G)

    def _add_to(self, x_GG):
        x_GG.flat[::self.nG + 1] += self.Vbare_G


class CoulombKernel:
    def __init__(self, truncation, *, N_c, pbc_c, kd):
        self.truncation = truncation
        assert self.truncation in {None, '0D', '2D'}
        self.N_c = N_c
        self.pbc_c = pbc_c
        self.kd = kd

    @classmethod
    def from_gs(cls, gs, *, truncation):
        return cls(truncation,
                   N_c=gs.kd.N_c,
                   pbc_c=gs.atoms.get_pbc(),
                   kd=gs.kd)

    def new(self, *, truncation):
        return CoulombKernel(truncation,
                             N_c=self.N_c,
                             pbc_c=self.pbc_c,
                             kd=self.kd)

    def description(self):
        if self.truncation is None:
            return 'No Coulomb truncation'
        else:
            return f'{self.truncation} Coulomb truncation'

    def sqrtV(self, qpd, q_v=None):
        return self.V(qpd, q_v=q_v)**0.5

    def V(self, qpd, q_v=None):
        assert isinstance(qpd, SingleQPWDescriptor)
        return get_coulomb_kernel(qpd, self.N_c, pbc_c=self.pbc_c, q_v=q_v,
                                  truncation=self.truncation)

    def kernel(self, qpd, q_v=None):
        return np.diag(self.V(qpd, q_v=q_v))

    def integrated_kernel(self, qpd, reduced, tofirstbz=False, *, N):
        return get_integrated_kernel(
            qpd=qpd, N_c=self.N_c, pbc_c=self.pbc_c,
            truncation=self.truncation, reduced=reduced,
            tofirstbz=tofirstbz, N=N)
```

```
def get_coulomb_kernel(qpd, N_c, q_v=None, truncation=None, *, pbc_c)\
        -> np.ndarray:
    """Factory function that calls the specified flavour
    of the Coulomb interaction"""

    if truncation is None:
        qG_Gv = qpd.get_reciprocal_vectors(add_q=True)
        if q_v is not None:
            assert qpd.optical_limit
            qG_Gv += q_v
        if qpd.optical_limit and q_v is None:
            v_G = np.zeros(len(qpd.G2_qG[0]))
            v_G[0] = 4 * np.pi
            v_G[1:] = 4 * np.pi / (qG_Gv[1:]**2).sum(axis=1)
        else:
            v_G = 4 * np.pi / (qG_Gv**2).sum(axis=1)

    elif truncation == '2D':
        v_G = calculate_2D_truncated_coulomb(qpd, pbc_c=pbc_c, q_v=q_v)
        if qpd.optical_limit and q_v is None:
            v_G[0] = 0.0

    elif truncation == '0D':
        from gpaw.hybrids.wstc import WignerSeitzTruncatedCoulomb
        wstc = WignerSeitzTruncatedCoulomb(qpd.gd.cell_cv, np.ones(3, int))
        v_G = wstc.get_potential(qpd)

    elif truncation in {'1D'}:
        raise feature_removed()

    else:
        raise ValueError('Truncation scheme %s not implemented' % truncation)

    return v_G.astype(complex)


def calculate_2D_truncated_coulomb(qpd, q_v=None, *, pbc_c):
    """ Simple 2D truncation of Coulomb kernel PRB 73, 205119.

        Note: q_v is only added to qG_Gv if qpd.optical_limit is True.

    """

    qG_Gv = qpd.get_reciprocal_vectors(add_q=True)
    if qpd.optical_limit:
        if q_v is not None:
            qG_Gv += q_v
        else:  # only to avoid warning. Later set to zero in factory function
            qG_Gv[0] = [1., 1., 1.]
    else:
        assert q_v is None

    # The non-periodic direction is determined from pbc_c
    Nn_c = np.where(~pbc_c)[0]
    Np_c = np.where(pbc_c)[0]
    assert len(Nn_c) == 1
    assert len(Np_c) == 2
    # Truncation length is half of cell vector in non-periodic direction
    R = qpd.gd.cell_cv[Nn_c[0], Nn_c[0]] / 2.

    qGp_G = ((qG_Gv[:, Np_c[0]])**2 + (qG_Gv[:, Np_c[1]]**2))**0.5
    qGn_G = qG_Gv[:, Nn_c[0]]

    v_G = 4 * np.pi / (qG_Gv**2).sum(axis=1)
    if np.allclose(qGn_G[0], 0) or qpd.optical_limit:
        """sin(qGn_G * R) = 0 when R = L/2 and q_n = 0.0"""
        v_G *= 1.0 - np.exp(-qGp_G * R) * np.cos(qGn_G * R)
    else:
```

```
        """Normal component of q is not zero"""
        a_G = qGn_G / qGp_G * np.sin(qGn_G * R) - np.cos(qGn_G * R)
        v_G *= 1. + np.exp(-qGp_G * R) * a_G

    return v_G.astype(complex)


def get_integrated_kernel(qpd, N_c, truncation=None,
                          reduced=False, tofirstbz=False, *, pbc_c, N):
    from scipy.special import j1, k0, j0, k1  # type: ignore
    # ignore type hints for the above import
    B_cv = 2 * np.pi * qpd.gd.icell_cv
    Nf_c = np.array([N, N, N])
    if reduced:
        # Only integrate periodic directions if truncation is used
        Nf_c[np.where(~pbc_c)[0]] = 1

    q_qc = monkhorst_pack(Nf_c)

    if tofirstbz:
        # We make a 1st BZ shaped integration volume, by reducing the full
        # Monkhorts-Pack grid to the 1st BZ.
        q_qc = to1bz(q_qc, qpd.gd.cell_cv)

    q_qc /= N_c
    q_qc += qpd.q_c

    if tofirstbz:
        # Because we added the q_c q-point vector, the integration volume is
        # no longer strictly inside the 1st BZ of the full cell. Thus, we
        # reduce it again.
        q_qc = to1bz(q_qc, qpd.gd.cell_cv)

    q_qv = np.dot(q_qc, B_cv)

    Nn_c = np.where(~pbc_c)[0]
    Np_c = np.where(pbc_c)[0]
    assert len(Nn_c) + len(Np_c) == 3

    if truncation is None:
        V_q = 4 * np.pi / np.sum(q_qv**2, axis=1)
        if len(Np_c) < 3:
            warnings.warn(f"You should be using truncation='{len(Np_c)}D'")
    elif truncation == '2D':
        assert len(Np_c) == 2

        # Truncation length is half of cell vector in non-periodic direction
        R = qpd.gd.cell_cv[Nn_c[0], Nn_c[0]] / 2.

        qp_q = ((q_qv[:, Np_c[0]])**2 + (q_qv[:, Np_c[1]]**2))**0.5
        qn_q = q_qv[:, Nn_c[0]]

        V_q = 4 * np.pi / (q_qv**2).sum(axis=1)
        a_q = qn_q / qp_q * np.sin(qn_q * R) - np.cos(qn_q * R)
        V_q *= 1. + np.exp(-qp_q * R) * a_q
    elif truncation == '1D':
        assert len(Np_c) == 1
        # The radius is determined from area of non-periodic part of cell
        Acell_cv = qpd.gd.cell_cv[Nn_c, :][:, Nn_c]
        R = abs(np.linalg.det(Acell_cv) / np.pi)**0.5

        qnR_q = (q_qv[:, Nn_c[0]]**2 + q_qv[:, Nn_c[1]]**2)**0.5 * R
        qpR_q = abs(q_qv[:, Np_c[0]]) * R
        V_q = 4 * np.pi / (q_qv**2).sum(axis=1)
        V_q *= (1.0 + qnR_q * j1(qnR_q) * k0(qpR_q)
                - qpR_q * j0(qnR_q) * k1(qpR_q))
    elif truncation == '0D':
        assert len(Np_c) == 0
        R = (3 * qpd.gd.volume / (4 * np.pi))**(1. / 3.)
```

```
        q2_q = (q_qv**2).sum(axis=1)
        V_q = 4 * np.pi / q2_q
        V_q *= 1.0 - np.cos(q2_q**0.5 * R)

    return np.sum(V_q) / len(V_q), np.sum(V_q**0.5) / len(V_q)


def feature_removed():
    return RuntimeError(
        '0D and 1D truncation have been removed due to not being tested.  '
        'If you need them, please find them in '
        'ec9e49e25613bb99cd69eec9d2613e38b9f6e6e1 '
        'and make sure to add tests in order to have them re-added.')
```