```python
"""XC density kernels for response function calculations."""
from abc import ABC, abstractmethod
from dataclasses import dataclass

import numpy as np

from gpaw.response import ResponseGroundStateAdapter, ResponseContext
from gpaw.response.pair_functions import SingleQPWDescriptor
from gpaw.response.localft import LocalFTCalculator
from gpaw.response.fxc_kernels import AdiabaticFXCCalculator


@dataclass
class DensityXCKernel(ABC):
    gs: ResponseGroundStateAdapter
    context: ResponseContext
    functional: str

    def __post_init__(self):
        assert self.gs.nspins == 1

    @staticmethod
    def from_functional(gs, context, functional, **kwargs):
        """Factory function creating DensityXCKernels.

        Choose between ALDA, Bootstrap and LRalpha (long-range kernel), where
        alpha is a user specified parameter (for example functional='LR0.25').
        """
        if functional[0] == 'A':
            return AdiabaticDensityKernel(gs, context, functional, kwargs)
        elif functional[:2] == 'LR':
            return LRDensityKernel(gs, context, functional)
        elif functional == 'Bootstrap':
            return BootstrapDensityKernel(gs, context, functional)
        raise ValueError(
            'Invalid functional for the density-density xc kernel:'
            f'{functional}')

    def __call__(self, qpd: SingleQPWDescriptor, chi0_wGG=None):
        self.context.print(f'Calculating {self}')
        return self.calculate(qpd=qpd, chi0_wGG=chi0_wGG)

    @abstractmethod
    def __repr__(self):
        """String representation of the density xc kernel."""

    @abstractmethod
    def calculate(self, *args, **kwargs):
        """Calculate the xc kernel.

        Since the exchange-correlation kernel is going to be rescaled according
        to the bare Coulomb interaction,

        K̃_xc(q) = v^(-1/2)(q) K_xc(q) v^(-1/2)(q)

        and that the Coulomb interaction in the optical q→0 limit leaves the
        long-range q-dependence out, see gpaw.response.coulomb_kernels,

        v(q) = 4π/|G+q| -> 4π for G==0 and 4π/|G| otherwise if q==0,

        we need to account to account for it here. That is,

        For q→0, the head and wings of the returned K_xc(q) needs to be
        rescaled according to K_xc(q) -> q K_xc(q) q
        """


@dataclass
class AdiabaticDensityKernel(DensityXCKernel):
    rshe_kwargs: dict

    def __post_init__(self):
        super().__post_init__()
        localft_calc = LocalFTCalculator.from_rshe_parameters(
            self.gs, self.context, **self.rshe_kwargs)
        self.fxc_calc = AdiabaticFXCCalculator(localft_calc)

    def __repr__(self):
        return f'{self.functional} kernel'

    def calculate(self, qpd: SingleQPWDescriptor, **ignored):
        fxc_kernel = self.fxc_calc(self.functional, '00', qpd)
        Kxc_GG = fxc_kernel.get_Kxc_GG()
        if qpd.optical_limit:
            Kxc_GG[0, :] = 0.0
            Kxc_GG[:, 0] = 0.0
        return Kxc_GG


@dataclass
class LRDensityKernel(DensityXCKernel):

    def __post_init__(self):
        super().__post_init__()
        self.alpha = float(self.functional[2:])

    def __repr__(self):
        return f'LR kernel with alpha = {self.alpha}'

    def calculate(self, qpd: SingleQPWDescriptor, **ignored):
        Kxc_sGG = calculate_lr_kernel(qpd, alpha=self.alpha)
        return Kxc_sGG[0]


@dataclass
class BootstrapDensityKernel(DensityXCKernel):

    def __repr__(self):
        return 'Bootstrap kernel'

    def calculate(self, qpd, *, chi0_wGG):
        Kxc_sGG = get_bootstrap_kernel(qpd, chi0_wGG, self.context)
        return Kxc_sGG[0]


def calculate_lr_kernel(qpd, alpha=0.2):
    """Long range kernel: fxc = \alpha / |q+G|^2"""

    assert qpd.optical_limit

    f_G = np.zeros(len(qpd.G2_qG[0]))
    f_G[0] = -alpha
    f_G[1:] = -alpha / qpd.G2_qG[0][1:]

    return np.array([np.diag(f_G)])


def get_bootstrap_kernel(qpd, chi0_wGG, context):
    """ Bootstrap kernel (see below) """

    if context.comm.rank == 0:
        chi0_GG = chi0_wGG[0]
        if context.comm.size > 1:
            # If size == 1, chi0_GG is not contiguous, and broadcast()
            # will fail in debug mode.  So we skip it until someone
            # takes a closer look.
            context.comm.broadcast(chi0_GG, 0)
    else:
        nG = qpd.ngmax
        chi0_GG = np.zeros((nG, nG), complex)
        context.comm.broadcast(chi0_GG, 0)

    return calculate_bootstrap_kernel(qpd, chi0_GG, context)


def calculate_bootstrap_kernel(qpd, chi0_GG, context):
    """Bootstrap kernel PRL 107, 186401"""
    p = context.print

    if qpd.optical_limit:
        v_G = np.zeros(len(qpd.G2_qG[0]))
        v_G[0] = 4 * np.pi
        v_G[1:] = 4 * np.pi / qpd.G2_qG[0][1:]
    else:
        v_G = 4 * np.pi / qpd.G2_qG[0]

    nG = len(v_G)
    K_GG = np.diag(v_G)

    Kxc_GG = np.zeros((nG, nG), dtype=complex)
    dminv_GG = np.zeros((nG, nG), dtype=complex)

    for iscf in range(120):
        dminvold_GG = dminv_GG.copy()
        Kxc_GG = K_GG + Kxc_GG

        chi_GG = np.dot(np.linalg.inv(np.eye(nG, nG)
                                      - np.dot(chi0_GG, Kxc_GG)), chi0_GG)
        dminv_GG = np.eye(nG, nG) + np.dot(K_GG, chi_GG)

        alpha = dminv_GG[0, 0] / (K_GG[0, 0] * chi0_GG[0, 0])
        Kxc_GG = alpha * K_GG
        p(iscf, 'alpha =', alpha, flush=False)
        error = np.abs(dminvold_GG - dminv_GG).sum()
        if np.sum(error) < 0.1:
            p('Self consistent fxc finished in %d iterations !' % iscf)
            break
        if iscf > 100:
            p('Too many fxc scf steps !')

    return np.array([Kxc_GG])
```