```python
from __future__ import annotations

from abc import ABC, abstractmethod
from collections.abc import Sequence
from dataclasses import dataclass

import numpy as np

from gpaw.response import timer
from gpaw.response.pair_functions import Chi


class HXCScaling(ABC):
    """Helper for scaling the hxc contribution to Dyson equations."""

    def __init__(self, lambd=None):
        self._lambd = lambd

    @property
    def lambd(self):
        return self._lambd

    def calculate_scaling(self, dyson_equations):
        self._lambd = self._calculate_scaling(dyson_equations)

    @abstractmethod
    def _calculate_scaling(self, dyson_equations: DysonEquations) -> float:
        """Calculate hxc scaling coefficient."""


class PWKernel(ABC):
    """Hartree, exchange and/correleation kernels in a plane-wave basis."""

    @property
    def nG(self):
        return self.get_number_of_plane_waves()

    def add_to(self, x_GG):
        assert x_GG.shape == (self.nG, self.nG)
        self._add_to(x_GG)

    @abstractmethod
    def get_number_of_plane_waves(self):
        """Return the size of the plane-wave basis."""

    @abstractmethod
    def _add_to(self, x_GG):
        """Add plane-wave kernel to an input array."""


class NoKernel(PWKernel):
    """A plane-wave kernel equal to zero."""

    def __init__(self, *, nG):
        self._nG = nG

    @classmethod
    def from_qpd(cls, qpd):
        qG_Gv = qpd.get_reciprocal_vectors(add_q=True)
        return cls(nG=qG_Gv.shape[0])

    def get_number_of_plane_waves(self):
        return self._nG

    def _add_to(self, x_GG):
        pass


@dataclass
```

```
class HXCKernel:
    """Hartree-exchange-correlation kernel in a plane-wave basis."""
    hartree_kernel: PWKernel
    xc_kernel: PWKernel

    def __post_init__(self):
        assert self.hartree_kernel.nG == self.xc_kernel.nG
        self.nG = self.hartree_kernel.nG

    def get_Khxc_GG(self):
        """Hartree-exchange-correlation kernel."""
        # Allocate array
        Khxc_GG = np.zeros((self.nG, self.nG), dtype=complex)
        self.hartree_kernel.add_to(Khxc_GG)
        self.xc_kernel.add_to(Khxc_GG)
        return Khxc_GG


class DysonSolver:
    """Class for inversion of Dyson-like equations."""

    def __init__(self, context):
        self.context = context

    @timer('Invert Dyson-like equations')
    def __call__(self, chiks: Chi, self_interaction: HXCKernel | Chi,
                 hxc_scaling: HXCScaling | None = None) -> Chi:
        """Solve the dyson equation and return the many-body susceptibility."""
        dyson_equations = self.get_dyson_equations(chiks, self_interaction)
        if hxc_scaling:
            if not hxc_scaling.lambd:  # calculate, if it hasn't been already
                hxc_scaling.calculate_scaling(dyson_equations)
            lambd = hxc_scaling.lambd
            self.context.print(r'Rescaling the self-enhancement function by a '
                               f'factor of λ={lambd}')
        self.context.print('Inverting Dyson-like equations')
        return dyson_equations.invert(hxc_scaling=hxc_scaling)

    @staticmethod
    def get_dyson_equations(chiks, self_interaction):
        if isinstance(self_interaction, HXCKernel):
            return DysonEquationsWithKernel(chiks, hxc_kernel=self_interaction)
        elif isinstance(self_interaction, Chi):
            return DysonEquationsWithXi(chiks, xi=self_interaction)
        else:
            raise ValueError(
                f'Invalid encoding of the self-interaction {self_interaction}')


class DysonEquations(Sequence):
    """Sequence of Dyson-like equations at different complex frequencies z."""

    def __init__(self, chiks: Chi):
        assert chiks.distribution == 'zGG' and\
            chiks.blockdist.fully_block_distributed, \
            "chiks' frequencies need to be fully distributed over world"
        self.chiks = chiks
        # Inherit basic descriptors from chiks
        self.qpd = chiks.qpd
        self.zd = chiks.zd
        self.zblocks = chiks.blocks1d
        self.spincomponent = chiks.spincomponent

    def __len__(self):
        return self.zblocks.nlocal

    def invert(self, hxc_scaling: HXCScaling | None = None) -> Chi:
        """Invert Dyson equations to obtain χ(z)."""
        # Scaling coefficient of the self-enhancement function
```

```
        lambd = hxc_scaling.lambd if hxc_scaling else None
        chi = self.chiks.new()
        for z, dyson_equation in enumerate(self):
            chi.array[z] = dyson_equation.invert(lambd=lambd)
        return chi


class DysonEquationsWithKernel(DysonEquations):
    def __init__(self, chiks: Chi, *, hxc_kernel: HXCKernel):
        # Check compatibility
        nG = hxc_kernel.nG
        assert chiks.array.shape[1:] == (nG, nG)
        # Initialize
        super().__init__(chiks)
        self.Khxc_GG = hxc_kernel.get_Khxc_GG()

    def __getitem__(self, z):
        chiks_GG = self.chiks.array[z]
        xi_GG = chiks_GG @ self.Khxc_GG
        return DysonEquation(chiks_GG, xi_GG)


class DysonEquationsWithXi(DysonEquations):
    def __init__(self, chiks: Chi, *, xi: Chi):
        # Check compatibility
        assert xi.distribution == 'zGG' and \
            xi.blockdist.fully_block_distributed
        assert chiks.spincomponent == xi.spincomponent
        assert np.allclose(chiks.zd.hz_z, xi.zd.hz_z)
        assert np.allclose(chiks.qpd.q_c, xi.qpd.q_c)
        # Initialize
        super().__init__(chiks)
        self.xi = xi

    def __getitem__(self, z):
        return DysonEquation(self.chiks.array[z], self.xi.array[z])


class DysonEquation:
    """Dyson equation at wave vector q and frequency z.

    The Dyson equation is given in plane-wave components as

    χ(q,z) = χ_KS(q,z) + Ξ(q,z) χ(q,z),

    where the self-enhancement function encodes the electron correlations
    induced by by the effective (Hartree-exchange-correlation) interaction:

    Ξ(q,z) = χ_KS(q,z) K_hxc(q,z)

    See [to be published] for more information.
    """

    def __init__(self, chiks_GG, xi_GG):
        self.nG = chiks_GG.shape[0]
        self.chiks_GG = chiks_GG
        self.xi_GG = xi_GG

    def invert(self, lambd: float | None = None):
        """Invert the Dyson equation (with or without a rescaling of Ξ).

        χ(q,z) = [1 - λ Ξ(q,z)]^(-1) χ_KS(q,z)
        """
        if lambd is None:
            lambd = 1.  # no rescaling
        enhancement_GG = np.linalg.inv(np.eye(self.nG) - lambd * self.xi_GG)
        return enhancement_GG @ self.chiks_GG
```