```
from abc import ABC, abstractmethod
from functools import cached_property
import numpy as np
from gpaw.response import timer
from scipy.spatial import Delaunay
from scipy.linalg.blas import zher

import gpaw.cgpaw as cgpaw
from gpaw.utilities.blas import rk, mmm
from gpaw.utilities.progressbar import ProgressBar
from gpaw.response.pw_parallelization import Blocks1D


class Integrand(ABC):
    @abstractmethod
    def matrix_element(self, point):
        ...

    @abstractmethod
    def eigenvalues(self, point):
        ...


def czher(alpha: float, x, A) -> None:
    """Hermetian rank-1 update of upper half of A.

    A += alpha * np.outer(x.conj(), x)

    """
    AT = A.T
    out = zher(alpha, x, 1, 1, 0, len(x), AT, 1)
    assert out is AT


class Integrator:
    def __init__(self, cell_cv, context, blockcomm, kncomm):
        """Baseclass for Brillouin zone integration and band summation.

        Simple class to calculate integrals over Brilloun zones
        and summation of bands.

        context: ResponseContext
        """
        self.vol = abs(np.linalg.det(cell_cv))
        self.context = context
        self.blockcomm = blockcomm
        self.kncomm = kncomm

    def mydomain(self, domain):
        from gpaw.response.pw_parallelization import Blocks1D
        # This function does the same as distribute_domain
        # but on a flat list and without all the fluff.
        #
        # In progress: Getting rid of distribute_domain(),
        blocks = Blocks1D(self.kncomm, len(domain))
        return [domain[i] for i in range(blocks.a, blocks.b)]

    def integrate(self, **kwargs):
        raise NotImplementedError


class PointIntegrator(Integrator):
    """Integrate brillouin zone using a broadening technique.

    The broadening technique consists of smearing out the
    delta functions appearing in many integrals by some factor
    eta. In this code we use Lorentzians."""

    def integrate(self, *, task, wd, domain, integrand, out_wxx):
```

```
        """Integrate a response function over bands and kpoints."""

        self.context.print('Integral kind:', task.kind)

        mydomain = self.mydomain(domain)

        prefactor = (2 * np.pi)**3 / self.vol / domain.nkpts
        out_wxx /= prefactor * self.kncomm.size

        # Sum kpoints
        # Calculate integrations weight
        pb = ProgressBar(self.context.fd)
        for _, point in pb.enumerate(mydomain):
            n_MG = integrand.matrix_element(point)
            if n_MG is None:
                continue
            deps_M = integrand.eigenvalues(point)

            task.run(wd, n_MG, deps_M, out_wxx)

        # We have divided the broadcasted sum from previous update by
        # kncomm.size, and thus here is will be back to its original value.
        # This is to prevent allocating an extra large array.
        self.kncomm.sum(out_wxx)

        if self.blockcomm.size == 1 and task.symmetrizable_unless_blocked:
            # Fill in upper/lower triangle also:
            nx = out_wxx.shape[1]
            il = np.tril_indices(nx, -1)
            iu = il[::-1]

            if isinstance(task, Hilbert):
                # XXX special hack since one of them wants the other
                # triangle.
                for out_xx in out_wxx:
                    out_xx[il] = out_xx[iu].conj()
            else:
                for out_xx in out_wxx:
                    out_xx[iu] = out_xx[il].conj()

        out_wxx *= prefactor


class IntegralTask(ABC):
    # Unique string for each kind of integral:
    kind = '(unset)'

    # Some integrals kinds like to calculate upper or lower half of the output
    # when nblocks==1.  In that case, this boolean signifies to the
    # integrator that the output array should be symmetrized.
    #
    # Actually: We don't gain anything much by doing this boolean
    # more systematically, since it's just Hermitian and Hilbert that need
    # it, and then one of the Tetrahedron types which is not compatible
    # anyway.  We should probably not do this.
    symmetrizable_unless_blocked = False

    @abstractmethod
    def run(self, wd, n_mG, deps_m, out_wxx):
        """Add contribution from one point to out_wxx."""


class GenericUpdate(IntegralTask):
    kind = 'response function'
    symmetrizable_unless_blocked = False

    def __init__(self, eta, blockcomm, eshift=None):
        self.eta = eta
        self.blockcomm = blockcomm
```

```
        self.eshift = eshift or 0.0

    # @timer('CHI_0 update')
    def run(self, wd, n_mG, deps_m, chi0_wGG):
        """Update chi."""

        deps_m += self.eshift * np.sign(deps_m)
        deps1_m = deps_m + 1j * self.eta
        deps2_m = deps_m - 1j * self.eta

        blocks1d = Blocks1D(self.blockcomm, chi0_wGG.shape[2])

        for omega, chi0_GG in zip(wd.omega_w, chi0_wGG):
            x_m = (1 / (omega + deps1_m) - 1 / (omega - deps2_m))
            if blocks1d.blockcomm.size > 1:
                nx_mG = n_mG[:, blocks1d.myslice] * x_m[:, np.newaxis]
            else:
                nx_mG = n_mG * x_m[:, np.newaxis]

            mmm(1.0, np.ascontiguousarray(nx_mG.T), 'N', n_mG.conj(), 'N',
                1.0, chi0_GG)


class Hermitian(IntegralTask):
    kind = 'hermitian response function'
    symmetrizable_unless_blocked = True

    def __init__(self, blockcomm, eshift=None):
        self.blockcomm = blockcomm
        self.eshift = eshift or 0.0

    # @timer('CHI_0 hermetian update')
    def run(self, wd, n_mG, deps_m, chi0_wGG):
        """If eta=0 use hermitian update."""
        deps_m += self.eshift * np.sign(deps_m)

        blocks1d = Blocks1D(self.blockcomm, chi0_wGG.shape[2])

        for w, omega in enumerate(wd.omega_w):
            if blocks1d.blockcomm.size == 1:
                x_m = np.abs(2 * deps_m / (omega.imag**2 + deps_m**2))**0.5
                nx_mG = n_mG.conj() * x_m[:, np.newaxis]
                rk(-1.0, nx_mG, 1.0, chi0_wGG[w], 'n')
            else:
                x_m = np.abs(2 * deps_m / (omega.imag**2 + deps_m**2))
                mynx_mG = n_mG[:, blocks1d.myslice] * x_m[:, np.newaxis]
                mmm(-1.0, mynx_mG, 'T', n_mG.conj(), 'N', 1.0, chi0_wGG[w])


class Hilbert(IntegralTask):
    kind = 'spectral function'
    symmetrizable_unless_blocked = True

    def __init__(self, blockcomm, eshift=None):
        self.blockcomm = blockcomm
        self.eshift = eshift or 0.0

    # @timer('CHI_0 spectral function update (new)')
    def run(self, wd, n_mG, deps_m, chi0_wGG):
        """Update spectral function.

        Updates spectral function A_wGG and saves it to chi0_wGG for
        later hilbert-transform."""

        deps_m += self.eshift * np.sign(deps_m)
        o_m = abs(deps_m)
        w_m = wd.get_floor_index(o_m)

        blocks1d = Blocks1D(self.blockcomm, chi0_wGG.shape[2])
```

```
        # Sort frequencies
        argsw_m = np.argsort(w_m)
        sortedo_m = o_m[argsw_m]
        sortedw_m = w_m[argsw_m]
        sortedn_mG = n_mG[argsw_m]

        index = 0
        while 1:
            if index == len(sortedw_m):
                break

            w = sortedw_m[index]
            startindex = index
            while 1:
                index += 1
                if index == len(sortedw_m):
                    break
                if w != sortedw_m[index]:
                    break

            endindex = index

            # Here, we have same frequency range w, for set of
            # electron-hole excitations from startindex to endindex.
            o1 = wd.omega_w[w]
            o2 = wd.omega_w[w + 1]
            p = np.abs(1 / (o2 - o1)**2)
            p1_m = np.array(p * (o2 - sortedo_m[startindex:endindex]))
            p2_m = np.array(p * (sortedo_m[startindex:endindex] - o1))

            if blocks1d.blockcomm.size > 1 and w + 1 < wd.wmax:
                x_mG = sortedn_mG[startindex:endindex, blocks1d.myslice]
                mmm(1.0,
                    np.concatenate((p1_m[:, None] * x_mG,
                                    p2_m[:, None] * x_mG),
                                   axis=1).T.copy(),
                    'N',
                    sortedn_mG[startindex:endindex].T.copy(),
                    'C',
                    1.0,
                    chi0_wGG[w:w + 2].reshape((2 * blocks1d.nlocal,
                                               blocks1d.N)))

            if blocks1d.blockcomm.size <= 1 and w + 1 < wd.wmax:
                x_mG = sortedn_mG[startindex:endindex]
                l_Gm = (p1_m[:, None] * x_mG).T.copy()
                r_Gm = x_mG.T.copy()
                mmm(1.0, r_Gm, 'N', l_Gm, 'C', 1.0, chi0_wGG[w])
                l_Gm = (p2_m[:, None] * x_mG).T.copy()
                mmm(1.0, r_Gm, 'N', l_Gm, 'C', 1.0, chi0_wGG[w + 1])


class Intraband(IntegralTask):
    kind = 'intraband'
    symmetrizable_unless_blocked = False

    # @timer('CHI_0 intraband update')
    def run(self, wd, vel_mv, deps_M, chi0_wvv):
        """Add intraband contributions"""
        # Intraband is a little bit special, we use neither wd nor deps_M

        for vel_v in vel_mv:
            x_vv = np.outer(vel_v, vel_v)
            chi0_wvv[0] += x_vv


class OpticalLimit(IntegralTask):
    kind = 'response function wings'
```

```
    symmetrizable_unless_blocked = False

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

    # @timer('CHI_0 optical limit update')
    def run(self, wd, n_mG, deps_m, chi0_wxvG):
        """Optical limit update of chi."""
        deps1_m = deps_m + 1j * self.eta
        deps2_m = deps_m - 1j * self.eta

        for w, omega in enumerate(wd.omega_w):
            x_m = (1 / (omega + deps1_m) - 1 / (omega - deps2_m))
            chi0_wxvG[w, 0] += np.dot(x_m * n_mG[:, :3].T, n_mG.conj())
            chi0_wxvG[w, 1] += np.dot(x_m * n_mG[:, :3].T.conj(), n_mG)


class HermitianOpticalLimit(IntegralTask):
    kind = 'hermitian response function wings'
    symmetrizable_unless_blocked = False

    # @timer('CHI_0 hermitian optical limit update')
    def run(self, wd, n_mG, deps_m, chi0_wxvG):
        """Optical limit update of hermitian chi."""
        for w, omega in enumerate(wd.omega_w):
            x_m = - np.abs(2 * deps_m / (omega.imag**2 + deps_m**2))
            chi0_wxvG[w, 0] += np.dot(x_m * n_mG[:, :3].T, n_mG.conj())
            chi0_wxvG[w, 1] += np.dot(x_m * n_mG[:, :3].T.conj(), n_mG)


class HilbertOpticalLimit(IntegralTask):
    kind = 'spectral function wings'
    symmetrizable_unless_blocked = False

    # @timer('CHI_0 optical limit hilbert-update')
    def run(self, wd, n_mG, deps_m, chi0_wxvG):
        """Optical limit update of chi-head and -wings."""

        for deps, n_G in zip(deps_m, n_mG):
            o = abs(deps)
            w = wd.get_floor_index(o)
            if w + 1 >= wd.wmax:
                continue
            o1, o2 = wd.omega_w[w:w + 2]
            if o > o2:
                continue
            else:
                assert o1 <= o <= o2, (o1, o, o2)

            p = 1 / (o2 - o1)**2
            p1 = p * (o2 - o)
            p2 = p * (o - o1)
            x_vG = np.outer(n_G[:3], n_G.conj())
            chi0_wxvG[w, 0, :, :] += p1 * x_vG
            chi0_wxvG[w + 1, 0, :, :] += p2 * x_vG
            chi0_wxvG[w, 1, :, :] += p1 * x_vG.conj()
            chi0_wxvG[w + 1, 1, :, :] += p2 * x_vG.conj()


class Point:
    def __init__(self, kpt_c, K, spin):
        self.kpt_c = kpt_c
        self.K = K
        self.spin = spin


class Domain:
    def __init__(self, kpts_kc, spins):
        self.kpts_kc = kpts_kc
```

```
        self.spins = spins

    @property
    def nkpts(self):
        return len(self.kpts_kc)

    @property
    def nspins(self):
        return len(self.spins)

    def __len__(self):
        return self.nkpts * self.nspins

    def __getitem__(self, num) -> Point:
        K = num // self.nspins
        return Point(self.kpts_kc[K], K,
                     self.spins[num % self.nspins])

    def tesselation(self):
        tesselation = KPointTesselation(self.kpts_kc)
        tesselated_domains = Domain(tesselation.bzk_kc, self.spins)
        return tesselation, tesselated_domains


class KPointTesselation:
    def __init__(self, kpts):
        self._td = Delaunay(kpts)

    @property
    def bzk_kc(self):
        return self._td.points

    @cached_property
    def simplex_volumes(self):
        volumes_s = np.zeros(self._td.nsimplex, float)
        for s in range(self._td.nsimplex):
            K_k = self._td.simplices[s]
            k_kc = self._td.points[K_k]
            volume = np.abs(np.linalg.det(k_kc[1:] - k_kc[0])) / 6.
            volumes_s[s] = volume
        return volumes_s

    def tetrahedron_weight(self, K, deps_k, omega_w):
        simplices_s = self.pts_k[K]
        W_w = np.zeros(len(omega_w), float)
        vol_s = self.simplex_volumes[simplices_s]
        cgpaw.tetrahedron_weight(
            deps_k, self._td.simplices, K, simplices_s, W_w, omega_w, vol_s)
        return W_w

    @cached_property
    def pts_k(self):
        pts_k = [[] for n in range(self.nkpts)]
        for s, K_k in enumerate(self._td.simplices):
            A_kv = np.append(self._td.points[K_k],
                             np.ones(4)[:, np.newaxis], axis=1)

            D_kv = np.append((A_kv[:, :-1]**2).sum(1)[:, np.newaxis],
                             A_kv, axis=1)
            a = np.linalg.det(D_kv[:, np.arange(5) != 0])

            if np.abs(a) < 1e-10:
                continue

            for K in K_k:
                pts_k[K].append(s)

        return [np.array(pts_k[k], int) for k in range(self.nkpts)]
```

```
    @property
    def nkpts(self):
        return self._td.npoints

    @cached_property
    def neighbours_k(self):
        return [np.unique(self._td.simplices[self.pts_k[k]])
                for k in range(self.nkpts)]


class TetrahedronIntegrator(Integrator):
    """Integrate brillouin zone using tetrahedron integration.

    Tetrahedron integration uses linear interpolation of
    the eigenenergies and of the matrix elements
    between the vertices of the tetrahedron."""

    @timer('Spectral function integration')
    def integrate(self, *, domain, integrand, wd, out_wxx, task):
        """Integrate response function.

        Assume that the integral has the
        form of a response function. For the linear tetrahedron
        method it is possible calculate frequency dependent weights
        and do a point summation using these weights."""

        tesselation, alldomains = domain.tesselation()
        mydomain = self.mydomain(alldomains)

        with self.context.timer('eigenvalues'):
            deps_tMk = None  # t for term

            for point in alldomains:
                deps_M = -integrand.eigenvalues(point)
                if deps_tMk is None:
                    deps_tMk = np.zeros([alldomains.nspins, *deps_M.shape,
                                         tesselation.nkpts], float)
                deps_tMk[point.spin, :, point.K] = deps_M

        # Calculate integrations weight
        pb = ProgressBar(self.context.fd)
        for _, point in pb.enumerate(mydomain):
            deps_Mk = deps_tMk[point.spin]
            teteps_Mk = deps_Mk[:, tesselation.neighbours_k[point.K]]
            n_MG = integrand.matrix_element(point)

            # Generate frequency weights
            i0_M, i1_M = wd.get_index_range(teteps_Mk.min(1), teteps_Mk.max(1))
            with self.context.timer('tetrahedron weight'):
                W_Mw = []
                for deps_k, i0, i1 in zip(deps_Mk, i0_M, i1_M):
                    W_w = tesselation.tetrahedron_weight(
                        point.K, deps_k, wd.omega_w[i0:i1])
                    W_Mw.append(W_w)

            task.run(n_MG, deps_Mk, W_Mw, i0_M, i1_M, out_wxx)

        self.kncomm.sum(out_wxx)

        if self.blockcomm.size == 1 and task.symmetrizable_unless_blocked:
            # Fill in upper/lower triangle also:
            nx = out_wxx.shape[1]
            il = np.tril_indices(nx, -1)
            iu = il[::-1]
            for out_xx in out_wxx:
                out_xx[il] = out_xx[iu].conj()


class HilbertTetrahedron:
```

```
    kind = 'spectral function'
    symmetrizable_unless_blocked = True

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

    def run(self, n_MG, deps_Mk, W_Mw, i0_M, i1_M, out_wxx):
        """Update output array with dissipative part."""
        blocks1d = Blocks1D(self.blockcomm, out_wxx.shape[2])

        for n_G, deps_k, W_w, i0, i1 in zip(n_MG, deps_Mk, W_Mw,
                                            i0_M, i1_M):
            if i0 == i1:
                continue

            for iw, weight in enumerate(W_w):
                if blocks1d.blockcomm.size > 1:
                    myn_G = n_G[blocks1d.myslice].reshape((-1, 1))
                    # gemm(weight, n_G.reshape((-1, 1)), myn_G,
                    #      1.0, out_wxx[i0 + iw], 'c')
                    mmm(weight, myn_G, 'N', n_G.reshape((-1, 1)), 'C',
                        1.0, out_wxx[i0 + iw])
                else:
                    czher(weight, n_G.conj(), out_wxx[i0 + iw])


class HilbertOpticalLimitTetrahedron:
    kind = 'spectral function wings'
    symmetrizable_unless_blocked = False

    def run(self, n_MG, deps_Mk, W_Mw, i0_M, i1_M, out_wxvG):
        """Update optical limit output array with dissipative part of the head
        and wings."""
        for n_G, deps_k, W_w, i0, i1 in zip(n_MG, deps_Mk, W_Mw,
                                            i0_M, i1_M):
            if i0 == i1:
                continue

            for iw, weight in enumerate(W_w):
                x_vG = np.outer(n_G[:3], n_G.conj())
                out_wxvG[i0 + iw, 0, :, :] += weight * x_vG
                out_wxvG[i0 + iw, 1, :, :] += weight * x_vG.conj()
```