```
from __future__ import annotations
from typing import Tuple

import numpy as np
from time import ctime
from abc import abstractmethod

from ase.units import Hartree

from gpaw.utilities.blas import mmmx

from gpaw.response import ResponseGroundStateAdapter, ResponseContext, timer
from gpaw.response.symmetry import QSymmetryAnalyzer, QSymmetryInput
from gpaw.response.symmetrize import BodySymmetryOperators
from gpaw.response.frequencies import ComplexFrequencyDescriptor
from gpaw.response.pw_parallelization import PlaneWaveBlockDistributor
from gpaw.response.matrix_elements import (PlaneWaveMatrixElementCalculator,
                                           NewPairDensityCalculator,
                                           TransversePairPotentialCalculator)
from gpaw.response.pair_integrator import PairFunctionIntegrator
from gpaw.response.pair_transitions import PairTransitions
from gpaw.response.pair_functions import SingleQPWDescriptor, Chi


class GeneralizedSuscetibilityCalculator(PairFunctionIntegrator):
    r"""Abstract calculator for generalized Kohn-Sham susceptibilities.

    For any combination of plane-wave matrix elements f(K) and g(K), one may
    define a generalized Kohn-Sham susceptibility in the Lehmann representation


                                 __  __   __
                          1      \   \    \
    x̄_KS,GG'^μν(q,ω+iη) = ―     /_  /_   /_   σ^μ_ss' σ^ν_s's (f_nks - f_n'k+qs')
                          V
                             k   n,n' s,s'
                                   f_(nks,n'k+qs')(G+q) g_(n'k+qs',nks)(-G'-q)
                                 x ―――――――――――――――――――――――――――――――――
                                          ħω - (ε_n'k+qs' - ε_nks) + iħη

    where σ^μ and σ^ν are Pauli matrices and the plane-wave matrix elements are
    defined in terms of some real local functional of the electron
    (spin-)density f[n](r):

    f_(nks,n'k+qs')(G+q) = <ψ_nks| e^-i(G+q)r f(r) |ψ_n'k+qs'>
    """

    def __init__(self, gs: ResponseGroundStateAdapter, context=None,
                 nblocks=1,
                 ecut=50, gammacentered=False,
                 nbands=None,
                 bandsummation='pairwise',
                 **kwargs):
        """Contruct the GeneralizedSuscetibilityCalculator

        Parameters
        ----------
        gs : ResponseGroundStateAdapter
        context : ResponseContext
        nblocks : int
            Distribute the chiks_zGG array into nblocks (where nblocks is a
            divisor of context.comm.size)
        ecut : float (or None)
            Plane-wave cutoff in eV
        gammacentered : bool
            Center the grid of plane waves around the Γ-point (or the q-vector)
        nbands : int
            Number of bands to include in the sum over states
        bandsummation : str
            Band summation strategy (does not change the result, but can affect
```

```
            the run-time).
            'pairwise': sum over pairs of bands
            'double': double sum over band indices.
        kwargs : see gpaw.response.pair_integrator.PairFunctionIntegrator
        """
        if context is None:
            context = ResponseContext()
        assert isinstance(context, ResponseContext)

        super().__init__(gs, context, nblocks=nblocks, **kwargs)

        self.ecut = None if ecut is None else ecut / Hartree  # eV to Hartree
        self.gammacentered = gammacentered
        self.nbands = nbands
        self.bandsummation = bandsummation

        mecalc1, mecalc2 = self.create_matrix_element_calculators()
        self.matrix_element_calc1 = mecalc1
        self.matrix_element_calc2 = mecalc2
        if mecalc2 is not mecalc1:
            assert not self.qsymmetry.time_reversal, \
                'Cannot make use of time-reversal symmetry for generalized ' \
                'susceptibilities with two different matrix elements'

    @abstractmethod
    def create_matrix_element_calculators(self) -> Tuple[
            PlaneWaveMatrixElementCalculator,
            PlaneWaveMatrixElementCalculator]:
        """Create the desired site matrix element calculators."""

    def calculate(self, spincomponent, q_c, zd) -> Chi:
        r"""Calculate χ_KS,GG'^μν(q,z), where z = ω + iη

        Parameters
        ----------
        spincomponent : str
            Spin component (μν) of the Kohn-Sham susceptibility.
            Currently, '00', 'uu', 'dd', '+-' and '-+' are implemented.
        q_c : list or np.array
            Wave vector in relative coordinates
        zd : ComplexFrequencyDescriptor
            Complex frequencies z to evaluate χ_KS,GG'^μν(q,z) at.
        """
        return self._calculate(*self._set_up_internals(spincomponent, q_c, zd))

    def _set_up_internals(self, spincomponent, q_c, zd,
                          distribution='GZg'):
        r"""Set up internal data objects."""
        assert isinstance(zd, ComplexFrequencyDescriptor)

        # Set up the internal plane-wave descriptor
        qpdi = self.get_pw_descriptor(q_c, internal=True)

        # Prepare to sum over bands and spins
        transitions = self.get_band_and_spin_transitions(
            spincomponent, nbands=self.nbands,
            bandsummation=self.bandsummation)

        self.context.print(self.get_info_string(
            qpdi, len(zd), spincomponent, len(transitions)))

        # Create data structure
        chiks = self.create_chiks(spincomponent, qpdi, zd, distribution)

        return chiks, transitions

    def _calculate(self, chiks: Chi, transitions: PairTransitions):
        r"""Integrate χ_KS according to the specified transitions."""
        self.context.print('Initializing the matrix element PAW corrections')
```

```
        self.matrix_element_calc1.initialize_paw_corrections(chiks.qpd)
        if self.matrix_element_calc2 is not self.matrix_element_calc1:
            self.matrix_element_calc2.initialize_paw_corrections(chiks.qpd)

        # Perform the actual integration
        symmetries = self._integrate(chiks, transitions)

        # Symmetrize chiks according to the symmetries of the ground state
        self.symmetrize(chiks, symmetries)

        # Map to standard output format
        chiks = self.post_process(chiks)

        return chiks

    def get_pw_descriptor(self, q_c, internal=False):
        """Get plane-wave descriptor for the wave vector q_c.

        Parameters
        ----------
        q_c : list or ndarray
            Wave vector in relative coordinates
        internal : bool
            When using symmetries, the actual calculation of chiks must happen
            using a q-centered plane wave basis. If internal==True, as it is by
            default, the internal plane wave basis (used in the integration of
            chiks.array) is returned, otherwise the external descriptor is
            returned, corresponding to the requested chiks.
        """
        q_c = np.asarray(q_c, dtype=float)
        gd = self.gs.gd

        # Update to internal basis, if needed
        if internal and self.gammacentered and not self.qsymmetry.disabled:
            # In order to make use of the symmetries of the system to reduce
            # the k-point integration, the internal code assumes a plane wave
            # basis which is centered at q in reciprocal space.
            gammacentered = False
            # If we want to compute the pair function on a plane wave grid
            # which is effectively centered in the gamma point instead of q, we
            # need to extend the internal ecut such that the q-centered grid
            # encompasses all reciprocal lattice points inside the gamma-
            # centered sphere.
            # The reduction to the global gamma-centered basis will then be
            # carried out as a post processing step.

            # Compute the extended internal ecut
            B_cv = 2.0 * np.pi * gd.icell_cv  # Reciprocal lattice vectors
            q_v = q_c @ B_cv
            ecut = get_ecut_to_encompass_centered_sphere(q_v, self.ecut)
        else:
            gammacentered = self.gammacentered
            ecut = self.ecut

        qpd = SingleQPWDescriptor.from_q(q_c, ecut, gd,
                                         gammacentered=gammacentered)

        return qpd

    def create_chiks(self, spincomponent, qpd, zd, distribution):
        """Create a new Chi object to be integrated."""
        assert distribution in ['GZg', 'ZgG']
        blockdist = PlaneWaveBlockDistributor(self.context.comm,
                                              self.blockcomm,
                                              self.intrablockcomm)
        return Chi(spincomponent, qpd, zd,
                   blockdist, distribution=distribution)

    @timer('Add integrand to χ_KS')
```

```
def add_integrand(self, kptpair, weight, chiks):
    r"""Add generalized susceptibility integrand for a given k-point pair.

    Calculates the relevant matrix elements and adds the susceptibility
    integrand to the output data structure for all relevant band and spin
    transitions of the given k-point pair, k -> k + q.

    Depending on the bandsummation parameter, the integrand of the
    generalized susceptibility is given by:

    bandsummation: double

                __
                \   σ^μ_ss' σ^ν_s's (f_nks - f_n'k's')
    (...)_k =   /  ------------------------------------ f_kt(G+q) g_kt^*(G'+q)
                --       ħz - (ε_n'k's' - ε_nks)
                t

    where f_kt(G+q) = f_nks,n'k's'(G+q) and k'=k+q up to a reciprocal wave
    vector.

    bandsummation: pairwise

                __ /
                \  |  σ^μ_ss' σ^ν_s's (f_nks - f_n'k's')
    (...)_k =   /  |  ------------------------------------
                -- |       ħz - (ε_n'k's' - ε_nks)
                t  \
                                                        \
                σ^μ_s's σ^ν_ss' (f_nks - f_n'k's')      |
       - δ_n'>n ------------------------------------     | f_kt(G+q) g_kt^*(G'+q)
                    ħz + (ε_n'k's' - ε_nks)             |
                                                        /

    The integrand is added to the output array chiks_x multiplied with the
    supplied kptpair integral weight.
    """
    # Calculate the matrix elements f_kt(G+q) and g_kt(G+q)
    matrix_element1 = self.matrix_element_calc1(kptpair, chiks.qpd)
    if self.matrix_element_calc2 is self.matrix_element_calc1:
        matrix_element2 = matrix_element1
    else:
        matrix_element2 = self.matrix_element_calc2(kptpair, chiks.qpd)
    # Calculate the temporal part of the integrand
    if chiks.spincomponent == '00' and self.gs.nspins == 1:
        weight = 2 * weight
    x_mytZ = get_temporal_part(chiks.spincomponent, chiks.zd.hz_z,
                               kptpair, self.bandsummation)
    x_tZ = kptpair.tblocks.all_gather(x_mytZ)

    self._add_integrand(
        matrix_element1, matrix_element2, x_tZ, weight, chiks)

def _add_integrand(self, matrix_element1, matrix_element2, x_tZ,
                   weight, chiks):
    r"""Add the generalized susceptibility integrand based on distribution.

    This entail performing a sum of transition t and an outer product
    in the plane-wave components G and G',

                __
                \
    (...)_k =   /  x_t^μν(ħz) f_kt(G+q) g_kt^*(G'+q)
                --
                t

    where x_t^μν(ħz) is the temporal part of x̄_KS,GG'^μν(q,ω+iη).
    """
    _add_integrand = self.get_add_integrand_method(chiks.distribution)
    _add_integrand(matrix_element1, matrix_element2, x_tZ, weight, chiks)
```

```
def get_add_integrand_method(self, distribution):
    """_add_integrand seletor."""
    if distribution == 'ZgG':
        _add_integrand = self._add_integrand_ZgG
    elif distribution == 'GZg':
        _add_integrand = self._add_integrand_GZg
    else:
        raise ValueError(f'Invalid distribution {distribution}')
    return _add_integrand

def _add_integrand_ZgG(self, matrix_element1, matrix_element2, x_tZ,
                       weight, chiks):
    """Add integrand in ZgG distribution.

    Z = global complex frequency index
    g = distributed G plane wave index
    G = global G' plane wave index
    """
    chiks_ZgG = chiks.array
    myslice = chiks.blocks1d.myslice

    with self.context.timer('Set up gcc and xf'):
        # Multiply the temporal part with the k-point integration weight
        x_Zt = np.ascontiguousarray(weight * x_tZ.T)

        # Set up f_kt(G+q) and g_kt^*(G'+q)
        f_tG = matrix_element1.get_global_array()
        if matrix_element2 is matrix_element1:
            g_tG = f_tG
        else:
            g_tG = matrix_element2.get_global_array()
        gcc_tG = g_tG.conj()

        # Set up x_t^μν(ħz) f_kt(G+q)
        f_gt = np.ascontiguousarray(f_tG[:, myslice].T)
        xf_Zgt = x_Zt[:, np.newaxis, :] * f_gt[np.newaxis, :, :]

    with self.context.timer('Perform sum over t-transitions of xf * gcc'):
        for xf_gt, chiks_gG in zip(xf_Zgt, chiks_ZgG):
            mmmx(1.0, xf_gt, 'N', gcc_tG, 'N', 1.0, chiks_gG)  # slow step

def _add_integrand_GZg(self, matrix_element1, matrix_element2, x_tZ,
                       weight, chiks):
    """Add integrand in GZg distribution.

    G = global G' plane wave index
    Z = global complex frequency index
    g = distributed G plane wave index
    """
    chiks_GZg = chiks.array
    myslice = chiks.blocks1d.myslice

    with self.context.timer('Set up gcc and xf'):
        # Multiply the temporal part with the k-point integration weight
        x_tZ *= weight

        # Set up f_kt(G+q) and g_kt^*(G'+q)
        f_tG = matrix_element1.get_global_array()
        if matrix_element2 is matrix_element1:
            g_tG = f_tG
        else:
            g_tG = matrix_element2.get_global_array()
        g_Gt = np.ascontiguousarray(g_tG.T)
        gcc_Gt = g_Gt.conj()

        # Set up x_t^μν(ħz) f_kt(G+q)
        f_tg = f_tG[:, myslice]
        xf_tZg = x_tZ[:, :, np.newaxis] * f_tg[:, np.newaxis, :]
```

```
        with self.context.timer('Perform sum over t-transitions of gcc * xf'):
            mmmx(1.0, gcc_Gt, 'N', xf_tZg, 'N', 1.0, chiks_GZg)  # slow step

    @timer('Symmetrizing chiks')
    def symmetrize(self, chiks, symmetries):
        """Symmetrize chiks_zGG."""
        operators = BodySymmetryOperators(symmetries, chiks.qpd)
        # Distribute over frequencies and symmetrize
        nz = len(chiks.zd)
        chiks_ZgG = chiks.array_with_view('ZgG')
        tmp_zGG = chiks.blockdist.distribute_as(chiks_ZgG, nz, 'zGG')
        operators.symmetrize_zGG(tmp_zGG)
        # Distribute over plane waves
        chiks_ZgG[:] = chiks.blockdist.distribute_as(tmp_zGG, nz, 'ZgG')

    @timer('Post processing')
    def post_process(self, chiks):
        """Cast a calculated chiks into a fixed output format."""
        if chiks.distribution != 'ZgG':
            # Always output chiks with distribution 'ZgG'
            chiks = chiks.copy_with_distribution('ZgG')

        if self.gammacentered and not self.qsymmetry.disabled:
            # Reduce the q-centered plane-wave basis used internally to the
            # gammacentered basis
            assert not chiks.qpd.gammacentered  # Internal qpd
            qpd = self.get_pw_descriptor(chiks.q_c)  # External qpd
            chiks = chiks.copy_with_reduced_pd(qpd)

        return chiks

    def get_info_string(self, qpd, nz, spincomponent, nt):
        r"""Get information about the χ_KS,GG'^μν(q,z) calculation"""
        from gpaw.utilities.memory import maxrss

        q_c = qpd.q_c
        ecut = qpd.ecut * Hartree
        Asize = nz * qpd.ngmax**2 * 16. / 1024**2 / self.blockcomm.size
        cmem = maxrss() / 1024**2

        isl = ['',
               'Setting up a generalized Kohn-Sham susceptibility calculation '
               'with:',
               f'    Spin component: {spincomponent}',
               f'    q_c: [{q_c[0]}, {q_c[1]}, {q_c[2]}]',
               f'    Number of frequency points: {nz}',
               self.get_band_and_transitions_info_string(self.nbands, nt),
               '',
               self.get_basic_info_string(),
               '',
               'Plane-wave basis of the generalized Kohn-Sham susceptibility:',
               f'    Planewave cutoff: {ecut}',
               f'    Number of planewaves: {qpd.ngmax}',
               '    Memory estimates:',
               f'        A_zGG: {Asize} M / cpu',
               f'        Memory usage before allocation: {cmem} M / cpu',
               '',
               f'{ctime()}']

        return '\n'.join(isl)


class ChiKSCalculator(GeneralizedSuscetibilityCalculator):
    r"""Calculator class for the four-component Kohn-Sham susceptibility tensor

    For collinear systems in absence of spin-orbit coupling,
    see [PRB 103, 245110 (2021)],
                              __  __   __
```

```
    χ_KS,GG'^μν(q,ω+iη) =  1  \   \    \
                           ‾  /   /    /   σ^μ_ss' σ^ν_s's (f_nks - f_n'k+qs')
                           V
                              k   n,n' s,s'
                                          n_nks,n'k+qs'(G+q) n_n'k+qs',nks(-G'-q)
                                     x ──────────────────────────────────────────
                                               ħω - (ε_n'k+qs' - ε_nks) + iħη

    where the matrix elements

    n_nks,n'k+qs'(G+q) = <nks| e^-i(G+q)r |n'k+qs'>

    are the plane-wave pair densities of each transition.
    """

    def create_matrix_element_calculators(self):
        pair_density_calc = NewPairDensityCalculator(self.gs, self.context)
        return pair_density_calc, pair_density_calc


class SelfEnhancementCalculator(GeneralizedSuscetibilityCalculator):
    r"""Calculator class for the transverse magnetic self-enhancement function.

    For collinear systems in absence of spin-orbit coupling,
    see [publication in preparation],

                          __  __   __
                       1  \   \    \
    Ξ_GG'^++(q,ω+iη) =  ‾  /   /    /   σ^+_ss' σ^-_s's (f_nks - f_n'k+qs')
                       V
                          k   n,n' s,s'
                                        n_nks,n'k+qs'(G+q) W^⊥_n'k+qs',nks(-G'-q)
                                   x ──────────────────────────────────────────
                                             ħω - (ε_n'k+qs' - ε_nks) + iħη

    where the matrix elements

    n_nks,n'k+qs'(G+q) = <nks| e^-i(G+q)r |n'k+qs'>

    and

    W^⊥_(nks,n'k+qs')(G+q) = <ψ_nks| e^-i(G+q)r f_LDA^-+(r) |ψ_n'k+qs'>

    are the plane-wave pair densities and transverse magnetic pair potentials
    respectively.
    """

    def __init__(self, gs: ResponseGroundStateAdapter, context=None,
                 rshelmax: int = -1,
                 rshewmin: float | None = None,
                 qsymmetry: QSymmetryInput = True,
                 **kwargs):
        """Construct the SelfEnhancementCalculator.

        Parameters
        ----------
        rshelmax : int
            The maximum index l (l < 6) to use in the expansion of the
            xc-kernel f_LDA^-+(r) into real spherical harmonics for the PAW
            correction.
        rshewmin : float or None
            If None, f_LDA^-+(r) will be fully expanded up to the chosen lmax.
            Given as a float (0 < rshewmin < 1), rshewmin indicates what
            coefficients to use in the expansion. If any (l,m) coefficient
            contributes with less than a fraction of rshewmin on average, it
            will not be included.
        """
        self.rshelmax = rshelmax
        self.rshewmin = rshewmin
```

```
        qsymmetry = QSymmetryAnalyzer.from_input(qsymmetry)
        super().__init__(gs, context=context,
                         qsymmetry=QSymmetryAnalyzer(
                             point_group=qsymmetry.point_group,
                             time_reversal=False),
                         **kwargs)

    def create_matrix_element_calculators(self):
        pair_density_calc = NewPairDensityCalculator(self.gs, self.context)
        pair_potential_calc = TransversePairPotentialCalculator(
            self.gs, self.context,
            rshelmax=self.rshelmax,
            rshewmin=self.rshewmin)
        return pair_density_calc, pair_potential_calc

    def _set_up_internals(self, spincomponent, *args, **kwargs):
        # For now, we are hardcoded to use the transverse pair potential,
        # calculating Ξ^++ corresponding to χ^+-
        assert spincomponent == '+-'
        return super()._set_up_internals(spincomponent, *args, **kwargs)


def get_ecut_to_encompass_centered_sphere(q_v, ecut):
    """Calculate the minimal ecut which results in a q-centered plane wave
    basis containing all the reciprocal lattice vectors G, which lie inside a
    specific gamma-centered sphere:

    |G|^2 < 2 * ecut
    """
    q = np.linalg.norm(q_v)
    ecut += q * (np.sqrt(2 * ecut) + q / 2)

    return ecut


def get_temporal_part(spincomponent, hz_z, kptpair, bandsummation):
    """Get the temporal part of a (causal linear) susceptibility, x_t^μν(ħz).
    """
    _get_temporal_part = create_get_temporal_part(bandsummation)
    return _get_temporal_part(spincomponent, hz_z, kptpair)


def create_get_temporal_part(bandsummation):
    """Creator component, deciding how to calculate the temporal part"""
    if bandsummation == 'double':
        return get_double_temporal_part
    elif bandsummation == 'pairwise':
        return get_pairwise_temporal_part
    raise ValueError(bandsummation)


def get_double_temporal_part(spincomponent, hz_z, kptpair):
    r"""Get:

                  σ^μ_ss' σ^ν_s's (f_nks - f_n'k's')
    x_t^μν(ħz) = ------------------------------------
                       ħz - (ε_n'k's' - ε_nks)
    """
    # Calculate σ^μ_ss' σ^ν_s's
    s1_myt, s2_myt = kptpair.get_local_spin_indices()
    scomps_myt = get_smat_components(spincomponent, s1_myt, s2_myt)
    # Calculate nominator
    nom_myt = - scomps_myt * kptpair.df_myt  # df = (f_n'k's' - f_nks)
    # Calculate denominator
    deps_myt = kptpair.deps_myt  # dε = (ε_n'k's' - ε_nks)
    denom_mytz = hz_z[np.newaxis] - deps_myt[:, np.newaxis]
    regularize_intraband_transitions(denom_mytz, kptpair)

    return nom_myt[:, np.newaxis] / denom_mytz
```

```
def get_pairwise_temporal_part(spincomponent, hz_z, kptpair):
    r"""Get:

                /
                | σ^μ_ss' σ^ν_s's (f_nks - f_n'k's')
    x_t^μν(ħz) = |
                |      ħz - (ε_n'k's' - ε_nks)
                \
                                                          \
                          σ^μ_s's σ^ν_ss' (f_nks - f_n'k's') |
                  - δ_n'>n ------------------------------- |
                                ħz + (ε_n'k's' - ε_nks)     |
                                                          /
    """
    # Kroenecker delta
    n1_myt, n2_myt = kptpair.get_local_band_indices()
    delta_myt = np.ones(len(n1_myt))
    delta_myt[n2_myt <= n1_myt] = 0
    # Calculate σ^μ_ss' σ^ν_s's and σ^μ_s's σ^ν_ss'
    s1_myt, s2_myt = kptpair.get_local_spin_indices()
    scomps1_myt = get_smat_components(spincomponent, s1_myt, s2_myt)
    scomps2_myt = get_smat_components(spincomponent, s2_myt, s1_myt)
    # Calculate nominators
    df_myt = kptpair.df_myt  # df = (f_n'k's' - f_nks)
    nom1_myt = - scomps1_myt * df_myt
    nom2_myt = - delta_myt * scomps2_myt * df_myt
    # Calculate denominators
    deps_myt = kptpair.deps_myt  # dε = (ε_n'k's' - ε_nks)
    denom1_mytz = hz_z[np.newaxis] - deps_myt[:, np.newaxis]
    denom2_mytz = hz_z[np.newaxis] + deps_myt[:, np.newaxis]
    regularize_intraband_transitions(denom1_mytz, kptpair)
    regularize_intraband_transitions(denom2_mytz, kptpair)

    return nom1_myt[:, np.newaxis] / denom1_mytz\
        - nom2_myt[:, np.newaxis] / denom2_mytz


def regularize_intraband_transitions(denom_mytx, kptpair):
    """Regularize the denominator of the temporal part in case of degeneracy.

    If the q-vector connects two symmetrically equivalent k-points inside a
    band, the occupation differences vanish and we regularize the denominator.

    NB: In principle there *should* be a contribution from the intraband
    transitions, but this is left for future work for now."""
    intraband_myt = kptpair.get_local_intraband_mask()
    degenerate_myt = np.abs(kptpair.deps_myt) < 1e-8
    denom_mytx[intraband_myt & degenerate_myt, ...] = 1.


def get_smat_components(spincomponent, s1_t, s2_t):
    """Calculate σ^μ_ss' σ^ν_s's for spincomponent (μν)."""
    smatmu = smat(spincomponent[0])
    smatnu = smat(spincomponent[1])
    return smatmu[s1_t, s2_t] * smatnu[s2_t, s1_t]


def smat(spinrot):
    if spinrot == '0':
        return np.array([[1, 0], [0, 1]])
    elif spinrot == 'u':
        return np.array([[1, 0], [0, 0]])
    elif spinrot == 'd':
        return np.array([[0, 0], [0, 1]])
    elif spinrot == '-':
        return np.array([[0, 0], [1, 0]])
    elif spinrot == '+':
```

```
        return np.array([[0, 1], [0, 0]])
    elif spinrot == 'z':
        return np.array([[1, 0], [0, -1]])
    else:
        raise ValueError(spinrot)
```