```
# materials.py is a subroutine of NumBAT that defines Material objects,
# these represent dispersive lossy refractive indices and possess
# methods to interpolate n from tabulated data.

# Copyright (C) 2017  Bjorn Sturmberg, Kokou Dossou.

# NumBAT is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.

# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.

# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.


import sys
import os
import traceback
import math
import json
import re
import copy
import numpy as np
#import numpy.linalg
import tempfile
import subprocess
#import scipy.linalg

import numbattools
from  nbtypes import unit_x, unit_y, unit_z
from bulkprops import *

import matplotlib as mpl
import matplotlib.cm as mplcm
import matplotlib.pyplot as plt
import matplotlib.colors as mplcolors
import matplotlib.ticker as ticker


from nbtypes import CrystalGroup
import reporting


class BadMaterialFileError(Exception):
    pass



# Array that converts between 4th rank tensors in terms of x,y,z and Voigt notat
ion
#                [[xx,xy,xz], [yx,yy,yz], [zx,zy,zz]]
to_Voigt = np.array([[0, 5, 4], [5, 1, 3], [4, 3, 2]])


g_material_library = None
```

```
def make_material(s):
    global g_material_library

    if g_material_library is None:
        g_material_library = MaterialLibrary()

    return g_material_library.get_material(s)


def rotate_tensor_elt(i, j, k, l, T_pqrs, mat_R):
    '''
  Calculates the element ijkl of the rotated tensor Tp from the original
  rank-4 tensor T_PQ in 6x6 Voigt notation under the rotation specified by the 3x3 matrix R.
  '''

    Tp_ijkl = 0

    for q in range(3):
        for r in range(3):
            V1 = to_Voigt[q, r]
            for s in range(3):
                for t in range(3):
                    V2 = to_Voigt[s, t]
                    Tp_ijkl += mat_R[i, q] * mat_R[j, r] * \
                        mat_R[k, s] * mat_R[l, t] * T_pqrs[V1, V2]

    return Tp_ijkl


def parse_rotation_axis(rot_axis_spec):
    '''Convert one of several forms - string, numpy 3vec -  to a standard unit 3vec'''

    if isinstance(rot_axis_spec, str):
        ral = rot_axis_spec.lower()
        if ral in ('x', 'x-axis'):
            rot_axis = unit_x
        elif ral in ('y', 'y-axis'):
            rot_axis = unit_y
        elif ral in ('z', 'z-axis'):
            rot_axis = unit_z
        else:
            reporting.report_and_exit(
                f"Can't convert {rot_axis_spec} to a 3-element unit vector.")
    else:  # should be a numeric 3 vector
        emsg = f"Can't convert {rot_axis_spec} to a 3-element unit vector."
        try:
            if isinstance(rot_axis_spec, (tuple, list)):  # try to convert to nu
mpy
                rot_axis = np.array(rot_axis_spec)
            elif isinstance(rot_axis_spec, np.ndarray):
                rot_axis = rot_axis_spec
            else:
                reporting.report_and_exit(emsg)
        except Exception:
            reporting.report_and_exit(emsg)
        if len(rot_axis) != 3:
            reporting.report_and_exit(
                f'Rotation axis {rot_axis} must have length 3.')

    nvec = np.linalg.norm(rot_axis)
    if numbattools.almost_zero(nvec):
        reporting.report_and_exit(f'Rotation axis {rot_axis} has zero length.')
```

```python
    return rot_axis/nvec


def _make_rotation_matrix(theta, rot_axis_spec):
    """
  Return the SO(3) matrix corresponding to a rotation of theta radians the specified rotation_axis.
  """

    uvec = parse_rotation_axis(rot_axis_spec)

    ct = math.cos(theta)
    st = math.sin(theta)
    omct = 1-ct
    ux, uy, uz = uvec[:]

    mat_R = np.array([
        [ct + ux**2*omct,  ux*uy*omct-uz*st, ux*uz*omct+uy*st],
        [uy*ux*omct+uz*st, ct + uy**2*omct,  uy*uz*omct-ux*st],
        [uz*ux*omct-uy*st, uz*uy*omct+ux*st, ct+uz**2*omct]
    ])

    reporting.assertion(numbattools.almost_unity(
        np.linalg.det(mat_R)), 'Rotation matrix has unit determinant.')

    return mat_R


def _rotate_3vector(vec, mat_R):
    # mat_R = _make_rotation_matrix(theta, rotation_axis)

    vrot = 0*vec
    for i in range(3):
        vrot[i] = mat_R[i, 0]*vec[0] + mat_R[i, 1]*vec[1] + mat_R[i, 2]*vec[2]

    return vrot


def _rotate_Voigt_tensor(T_PQ, mat_R):
    """
  Rotate an acoustic material tensor by theta radians around a specified rotation_axis.
  T_PQ is a rank-4 tensor expressed in 6x6 Voigt notation.

  The complete operation in 3x3x3x3 notation is
  T'_ijkl  = sum_{pqrs} R_ip R_jq R_kr R_ls  T_pqrs.

  The result T'_ijkl is returned in Voigt format T'_PQ.

  Args:
    T_PQ  (array): Tensor to be rotated.

    theta  (float): Angle to rotate by in radians.

    rotation_axis  (str): Axis around which to rotate.
  """

    # mat_R = _make_rotation_matrix(theta, rotation_axis)

    Tp_PQ = np.zeros((6, 6))
    for i in range(3):
        for j in range(3):
            V1 = to_Voigt[i, j]
```

```python
            for k in range(3):
                for l in range(3):
                    V2 = to_Voigt[k, l]
                    Tp_PQ[V1, V2] = rotate_tensor_elt(i, j, k, l, T_PQ, mat_R)

    return Tp_PQ



class VoigtTensor4(object):
    '''A class for representing rank 4 tensors in the compact Voigt representation.'''

    def __init__(self, material_name, symbol, src_dict=None):
        self.mat = np.zeros([7, 7], dtype=float)  # unit indexing
        self.material_name = material_name
        self.symbol = symbol  # eg 'c', 'p', 'eta'
        self.d = src_dict

    # Allow direct indexing of Voigt tensor in [(i,j)] form

    def __getitem__(self, k):
        return self.mat[k[0], k[1]]

    def __setitem__(self, k, v):
        self.mat[k[0], k[1]] = v

    def __str__(self):

        prec = np.get_printoptions()['precision']
        np.set_printoptions(precision=4)

        s = f'\nVoigt tensor {self.material_name}, tensor {self.symbol}:\n'
        s += str(self.mat[1:, 1:])

        np.set_printoptions(precision=prec)

        return s

    def dump(self):
        print(f'\nVoigt tensor {self.material_name}, tensor {self.symbol}')
        print(self.mat[1:, 1:])

    def as_zerobase_matrix(self):
        '''Returns copy of Voigt matrix indexed as m[0..5, 0..5].'''
        return self.mat[1:, 1:].copy()

    def read(self, m, n, optional=False):
        elt = f'{self.symbol}_{m}{n}'

        if elt not in self.d:
            if not optional:
                reporting.report_and_exit(
                    f'Failed to read required tensor element {elt} for material {self.material_name}')
            else:
                return False

        self.mat[m, n] = self.d[elt]
        return True

    def load_isotropic(self):
        self.read(1, 1)
        self.read(1, 2)
        self.read(4, 4)
        self.set_isotropic(self.mat[1, 1], self.mat[1, 2], self.mat[4, 4])

    def set_isotropic(self, m11, m12, m44):
        '''Build Voigt matrix from 3 parameters for isotropic geometry.
    (Actually, only two are independent.)'''

        self.mat[1, 1] = m11
        self.mat[1, 2] = m12
        self.mat[4, 4] = m44

        self.mat[2, 2] = self.mat[1, 1]
        self.mat[3, 3] = self.mat[1, 1]
        self.mat[5, 5] = self.mat[4, 4]
        self.mat[6, 6] = self.mat[4, 4]
        self.mat[2, 1] = self.mat[1, 2]
        self.mat[2, 3] = self.mat[1, 2]
        self.mat[1, 3] = self.mat[1, 2]
        self.mat[3, 1] = self.mat[1, 2]
        self.mat[3, 2] = self.mat[1, 2]

    def check_symmetries(self, sym=None):
        # Check matrix is symmetric and positive definite

        rtol = 1e-12
        tol = rtol * np.abs(self.mat).max()
        tmat = self.mat - self.mat.T
        mat_is_sym = numbattools.almost_zero(np.linalg.norm(tmat), tol)
        reporting.assertion(
            mat_is_sym, f'Material matrix {self.material_name}-{self.symbol} is symmetric.\n' + str
(self.mat))

    def rotate(self, matR):
        '''Rotates the crystal according to the SO(3) matrix matR.
    '''

        rot_tensor = _rotate_Voigt_tensor(self.mat[1:, 1:], matR)
        self.mat[1:, 1:] = rot_tensor


class MaterialLibrary:

    def __init__(self):

        self._data_loc = ''
        self._materials = {}

        # identify mat data directory:  backend/material_data
        this_dir = os.path.dirname(os.path.realpath(__file__))
        self._data_loc = os.path.join(this_dir, "material_data", "")

        self._load_materials()

    def get_material(self, matname):
        try:
            mat = self._materials[matname]
        except KeyError:
            reporting.report_and_exit(
                f'Material {matname} not found in material_data folder.\nEither the material file is missing or
 the name field in the material file has been incorrectly specified.')
```

```
        return mat

    def _load_materials(self):
        for fname in os.listdir(self._data_loc):
            if not fname.endswith(".json"):
                continue

            json_data = None
            with open(self._data_loc + fname, 'r') as fin:
                s_in = ''.join(fin.readlines())
                s_in = re.sub(r'//.*\n', '\n', s_in)

                try:
                    json_data = json.loads(s_in)
                except Exception as err:
                    traceback.print_exc()
                    reporting.report_and_exit(
                        f'JSON parsing error: {err} for file {self.json_file}')

            try:
                new_mat = Material(json_data, fname)
            except BadMaterialFileError as err:
                reporting.report_and_exit(str(err))

            if new_mat.material_name in self._materials:
                reporting.report_and_exit(
                    f"Material file {fname} has the same name as an existing material {new_mat.material_n
ame}.")

            self._materials[new_mat.material_name] = new_mat


class Material(object):
    """Class representing a waveguide material.

    This should not be constructed directly but by calling materials.get_material()

     Materials include the following properties and corresponding units:
      -  Refractive index []
      -  Density [kg/m3]
      -  Stiffness tensor component [Pa]
      -  Photoelastic tensor component []
      -  Acoustic loss tensor component [Pa s]

    """

    def __init__(self, json_data, filename):

        # a,b,c crystal axes according to standard conventions
        self._crystal_axes = []

        self.c_tensor = None
        self.eta_tensor = None
        self.p_tensor = None

        self._parse_json_data(json_data, filename)

    def __str__(self):
        s = (f'Material: {self.chemical}\n'
             f'  File: {self.material_name}\n'
             f'  Source: {self.author}\n'
             f'  Date: {self.date}' )
```

```python
        if len(self.comment):
            s += f'\nComment: {self.comment}'
        return s

    def copy(self):
        return copy.deepcopy(self)

    def full_str(self):
        s = str(self)
        s += str(self.c_tensor)
        s += str(self.eta_tensor)
        s += str(self.p_tensor)
        return s

    def elastic_properties(self):
        '''Returns a string containing key elastic properties of the material.'''

        dent = '\n '
        try:
            s = f'Elastic properties of material {self.material_name}'
            s += dent + f'Density:       {self.rho:.3f} kg/m^3'
            s += dent + f'Ref. index:    {self.refindex_n:.4f} '

            s += dent + f'Crystal class:  {self.crystal.name}'

            if self.is_isotropic():
                s += dent + f'c11:         {self.c_tensor.mat[1, 1]*1e−9:.3f} GPa'
                s += dent + f'c12:         {self.c_tensor.mat[1, 2]*1e−9:.3f} GPa'
                s += dent + f'c44:         {self.c_tensor.mat[4, 4]*1e−9:.3f} GPa'
                s += dent + f"Young's mod E:  {self.EYoung*1e−9:.3f} GPa"
                s += dent + f'Poisson ratio:  {self.nuPoisson:.3f}'
                s += dent + f'Velocity long.: {self.Vac_longitudinal():.3f} m/s'
                s += dent + f'Velocity shear: {self.Vac_shear():.3f} m/s'
            else:
                s += dent + 'Stiffness c_IJ:' + str(self.c_tensor) + '\n'

                # find wave properties for z propagation
                v_phase, v_evecs, v_vgroup = solve_christoffel(unit_z, self.c_te
nsor, self.rho)

                with np.printoptions(precision=4, floatmode='fixed', sign=' ', sup
press=True):
                    for m in range(3):
                        vgabs = np.linalg.norm(v_vgroup[m])
                        s += dent + f'Wave mode {m+1}: v_p={v_phase[m]:.4f} km/s,  |v_g|={vga
bs:.4f} km/s, ' \
                            + 'u_j=' + str(v_evecs[:,m]) + ', v_g=' + str(v_vgroup
[m]) +' km/s'


        except Exception:
            s = 'Unknown/undefined elastic parameters in material '+self.material_name
        return s

    def Vac_longitudinal(self):
        '''For an isotropic material, returns the longitudinal (P−wave) elastic phase velocity.'''
        assert (self.is_isotropic())

        if not self.rho or self.rho == 0:  # Catch vacuum cases
            return 0.
        else:
            return math.sqrt(self.c_tensor[1, 1]/self.rho)
```

```python

    def Vac_shear(self):
        '''For an isotropic material, returns the shear (S−wave) elastic phase velocity.'''
        assert (self.is_isotropic())

        if not self.rho or self.rho == 0:  # Catch vacuum cases
            return 0.
        else:
            return math.sqrt(self.c_tensor[4, 4]/self.rho)

    def has_elastic_properties(self):
        '''Returns true if the material has at least some elastic properties defined.'''
        return self.rho is not None

    def _parse_json_data(self, json_data, fname):
        """
Load material data from json file.

Args:
  data_file  (str): name of data file located in NumBAT/backend/material_data

"""
        self._params = json_data  # Do without this?

        # Name of this file, will be used as identifier and must be present
        self.material_name = json_data.get('material_name', 'NOFILENAME')
        if self.material_name == 'NOFILENAME':
            raise BadMaterialFileError(
                f"Material file {fname} has no 'material_name' field.")

        self.format = json_data.get('format', 'NOFORMAT')
        if self.format == 'NOFORMAT':
            raise BadMaterialFileError(
                f"Material file {fname} has no 'format' field.")

        if self.format != 'NumBATMaterial−fmt−2.0':
            raise BadMaterialFileError(
                f"Material file {fname} must be in format 'NumBATMaterial−Fmt−2.0'.")

        self.chemical = json_data['chemical']  # Chemical composition
        self.author = json_data['author']  # Author of data
        # Year of data publication/measurement
        self.date = json_data['date']
        # Source institution
        self.institution = json_data['institution']
        # doi or, failing that, the http address
        self.doi = json_data['doi']

        # general comment for any purpose
        self.comment = json_data.get('comment', '')

        Re_n = json_data['Re_n']  # Real part of refractive index []
        # Imaginary part of refractive index []
        Im_n = json_data['Im_n']
        self.refindex_n = (Re_n + 1j*Im_n)  # Complex refractive index []
        self.rho = json_data['s']  # Density [kg/m3]

        if self.is_vacuum():  # no mechanical properties available
            return

        self.EYoung = None
        self.nuPoisson = None
        if not 'crystal_class' in json_data:
            raise BadMaterialFileError(
                f"Material file {fname} has no 'crystal_class' field.")
        try:
            self.crystal = CrystalGroup[json_data['crystal_class']]
        except ValueError as exc:
            print('Unknown crystal class in material data file')
            raise BadMaterialFileError(
                f"Unknown crystal class in material data file {fname}") from exc

        if self.crystal == CrystalGroup.Isotropic:
            self.construct_crystal_isotropic()

        else:
            self.c_tensor = VoigtTensor4(self.material_name, 'c', json_data)
            self.eta_tensor = VoigtTensor4(
                self.material_name, 'eta', json_data)
            self.p_tensor = VoigtTensor4(self.material_name, 'p', json_data)
            # self.load_tensors()
            self.construct_crystal_anisotropic()
        self._store_original_tensors()

    def _store_original_tensors(self):
        self._c_tensor_orig = self.c_tensor
        self._p_tensor_orig = self.p_tensor
        self._eta_tensor_orig = self.eta_tensor

    def is_vacuum(self):
        '''Returns True if the material is the vacuum.'''
        return self.chemical == 'Vacuum'

    # (don't really need this as isotropic materials are the same)
    def construct_crystal_cubic(self):

        # plain cartesian axes
        self.set_crystal_axes(unit_x, unit_y, unit_z)

        try:
            self.c_tensor.read(1, 1)
            self.c_tensor.read(1, 2)
            self.c_tensor[1, 3] = self.c_tensor[1, 2]
            self.c_tensor[2, 1] = self.c_tensor[1, 2]
            self.c_tensor[2, 2] = self.c_tensor[1, 1]
            self.c_tensor[2, 3] = self.c_tensor[1, 2]
            self.c_tensor[3, 1] = self.c_tensor[1, 2]
            self.c_tensor[3, 2] = self.c_tensor[1, 2]
            self.c_tensor[3, 3] = self.c_tensor[1, 1]
            self.c_tensor.read(4, 4)
            self.c_tensor[5, 5] = self.c_tensor[4, 4]
            self.c_tensor[6, 6] = self.c_tensor[4, 4]

            self.eta_tensor.read(1, 1)
            self.eta_tensor.read(1, 2)
            self.eta_tensor[1, 3] = self.eta_tensor[1, 2]
            self.eta_tensor[2, 1] = self.eta_tensor[1, 2]
            self.eta_tensor[2, 2] = self.eta_tensor[1, 1]
            self.eta_tensor[2, 3] = self.eta_tensor[1, 2]
            self.eta_tensor[3, 1] = self.eta_tensor[1, 2]
            self.eta_tensor[3, 2] = self.eta_tensor[1, 2]
            self.eta_tensor[3, 3] = self.eta_tensor[1, 1]
            self.eta_tensor.read(4, 4)
```

```
            self.eta_tensor[5, 5] = self.eta_tensor[4, 4]
            self.eta_tensor[6, 6] = self.eta_tensor[4, 4]

            self.p_tensor.read(1, 1)
            self.p_tensor.read(1, 2)

            self.p_tensor[1, 3] = self.p_tensor[1, 2]
            self.p_tensor[2, 1] = self.p_tensor[1, 2]
            self.p_tensor[2, 2] = self.p_tensor[1, 1]
            self.p_tensor[2, 3] = self.p_tensor[1, 2]
            self.p_tensor[3, 1] = self.p_tensor[1, 2]
            self.p_tensor[3, 2] = self.p_tensor[1, 2]
            self.p_tensor[3, 3] = self.p_tensor[1, 1]
            self.p_tensor.read(4, 4)

            # According to Powell, for Oh group, these are distinct elements, bu
t no one seems to quote them
            if not self.p_tensor.read(5, 5, optional=True):
                self.p_tensor[5, 5] = self.p_tensor[4, 4]
            if not self.p_tensor.read(6, 6, optional=True):
                self.p_tensor[6, 6] = self.p_tensor[4, 4]

        except Exception:
            reporting.report_and_exit(
                f'Failed to load cubic crystal class in material data file {self.json_file}')

    def construct_crystal_trigonal(self):
        # Good source for these rules is the supp info of doi:10.1364/JOSAB.4826
56 (Gustavo surface paper)

        self.set_crystal_axes(unit_x, unit_y, unit_z)

        try:
            for lintens in [self.c_tensor, self.eta_tensor]:
                for (i, j) in [(1, 1), (1, 2), (1, 3), (1, 4), (3, 3), (4, 4)]:
                    lintens.read(i, j)

                lintens[2, 1] = lintens[1, 2]
                lintens[2, 2] = lintens[1, 1]
                lintens[2, 3] = lintens[1, 3]
                lintens[2, 4] = -lintens[1, 4]

                lintens[3, 1] = lintens[1, 3]
                lintens[3, 2] = lintens[1, 3]

                lintens[4, 1] = lintens[1, 4]
                lintens[4, 2] = -lintens[1, 4]

                lintens[5, 5] = lintens[4, 4]
                lintens[5, 6] = lintens[1, 4]
                lintens[6, 5] = lintens[1, 4]
                lintens[6, 6] = (lintens[1, 1]-lintens[1, 2])/2.0

            # TODO: confirm correct symmetry properties for p.
            # PreviouslyuUsing trigonal = C3v from Powell, now the paper above
            self.p_tensor.read(1, 1)
            self.p_tensor.read(1, 2)
            self.p_tensor.read(1, 3)
            self.p_tensor.read(1, 4)
            self.p_tensor.read(3, 1)
            self.p_tensor.read(3, 3)
            self.p_tensor.read(4, 1)
            self.p_tensor.read(4, 4)

            self.p_tensor[2, 1] = self.p_tensor[1, 2]
            self.p_tensor[2, 2] = self.p_tensor[1, 1]
            self.p_tensor[2, 3] = self.p_tensor[1, 3]
            self.p_tensor[2, 4] = -self.p_tensor[1, 4]

            self.p_tensor[3, 2] = self.p_tensor[3, 1]

            self.p_tensor[4, 2] = -self.p_tensor[4, 1]

            self.p_tensor[5, 5] = self.p_tensor[4, 4]
            self.p_tensor[5, 6] = self.p_tensor[4, 1]
            self.p_tensor[6, 5] = self.p_tensor[1, 4]
            self.p_tensor[6, 6] = (self.p_tensor[1, 1] - self.p_tensor[1, 2])/2

        except Exception:
            reporting.report_and_exit(
                f'Failed to load trigonal crystal class in material data file {self.json_file}')

    def construct_crystal_general(self):
        try:  # full anisotropic tensor components
            for i in range(1, 7):
                for j in range(1, 7):
                    self.c_tensor.read(i, j)
                    self.p_tensor.read(i, j)
                    self.eta_tensor.read(i, j)

        except KeyError:
            reporting.report_and_exit(
                'Failed to load anisotropic crystal class in material data file {self.json_file}')

    def set_refractive_index(self, nr, ni=0.0):
        self.refindex_n = nr + 1j*ni

    def is_isotropic(self): return not self._anisotropic

    # deprecated
    def rotate_axis(self, theta, rotation_axis, save_rotated_tensors=False):
        reporting.register_warning(
            'rotate_axis function is depprecated. Use rotate()')
        self.rotate(theta, rotation_axis, save_rotated_tensors)

    def rotate(self, theta, rot_axis_spec, save_rotated_tensors=False):
        """ Rotate crystal axis by theta radians.

          Args:
              theta  (float): Angle to rotate by in radians.

              rotate_axis  (str): Axis around which to rotate.

          Keyword Args:
              save_rotated_tensors  (bool): Save rotated tensors to csv.

          Returns:
              ``Material`` object with rotated tensor values.
    """

        rotation_axis = parse_rotation_axis(rot_axis_spec)
        matR = _make_rotation_matrix(theta, rotation_axis)

        self.c_tensor.rotate(matR)
```

```
        self.p_tensor.rotate(matR)
        self.eta_tensor.rotate(matR)

        self.c_tensor.check_symmetries()

        caxes = self._crystal_axes.copy()
        self.set_crystal_axes(
            _rotate_3vector(caxes[0], matR),
            _rotate_3vector(caxes[1], matR),
            _rotate_3vector(caxes[2], matR)
        )

        if save_rotated_tensors:
            np.savetxt('rotated_c_tensor.csv',
                       self.c_tensor.mat, delimiter=',')
            np.savetxt('rotated_p_tensor.csv',
                       self.p_tensor.mat, delimiter=',')
            np.savetxt('rotated_eta_tensor.csv',
                       self.eta_tensor.mat, delimiter=',')

    # restore orientation to original axes in spec file.
    def reset_orientation(self):

        self.c_tensor = copy.deepcopy(self._c_tensor_orig)
        self.p_tensor = copy.deepcopy(self._p_tensor_orig)
        self.eta_tensor = copy.deepcopy(self._eta_tensor_orig)

        self.set_crystal_axes(unit_x, unit_y, unit_z)

    # rotate original crystal to specific named-orientation, eg x-cut, y-cut. '1
11' etc.
    def set_orientation(self, label):
        self.reset_orientation()

        try:
            ocode = self._params[f'orientation_{label.lower()}']
        except KeyError:
            reporting.report_and_exit(
                f'Orientation "{label}" is not defined for material {self.material_name}.')

        if ocode == 'ident':  # native orientation is the desired one
            return

        try:
            ux, uy, uz, rot = map(float, ocode.split(','))
        except:
            reporting.report_and_exit(
                f"Can't parse crystal orientation code {ocode} for material {self.material_name}.")
        rot_axis = np.array((ux, uy, uz))
        theta = rot*np.pi/180

        self.rotate(theta, rot_axis)

    def set_crystal_axes(self, va, vb, vc):
        self._crystal_axes = [va, vb, vc]

    def construct_crystal_isotropic(self):
        # ordinary Cartesian axes for the crystal axes
        self.set_crystal_axes(unit_x, unit_y, unit_z)

        self._anisotropic = False
        # Try to read isotropic from stiffness and then from Young's modulus and
 Poisson ratio
        if 'c_11' in self._params and 'c_12' in self._params and 'c_44' in self._p
arams:
            self.c_tensor = VoigtTensor4(self.material_name, 'c', self._params)
            self.c_tensor.load_isotropic()
            mu = self.c_tensor.mat[4, 4]
            lam = self.c_tensor.mat[1, 2]
            r = lam/mu
            self.nuPoisson = 0.5*r/(1+r)
            self.EYoung = 2*mu*(1+self.nuPoisson)

        elif 'EYoung' in self._params and 'nuPoisson' in self._params:
            self.EYoung = self._params['EYoung']
            self.nuPoisson = self._params['nuPoisson']
            c44 = 0.5*self.EYoung/(1+self.nuPoisson)
            c12 = self.EYoung*self.nuPoisson / \
                ((1+self.nuPoisson) * (1-2*self.nuPoisson))
            c11 = c12+2*c44
            self.c_tensor = VoigtTensor4(self.material_name, 'c')
            self.c_tensor.set_isotropic(c11, c12, c44)
        else:
            reporting.report_and_exit(
                'Broken isotropic material file:' + self.json_file)

        self.eta_tensor = VoigtTensor4(self.material_name,
                                       'eta', self._params)
        self.p_tensor = VoigtTensor4(self.material_name, 'p', self._params)

        self.p_tensor.load_isotropic()
        self.eta_tensor.load_isotropic()

        self.c_tensor.check_symmetries()

    # not do this unless symmetry is off?
    def construct_crystal_anisotropic(self):

        self.c_tensor = VoigtTensor4(self.material_name, 'c', self._params)
        self.eta_tensor = VoigtTensor4(self.material_name, 'eta', self._params)
        self.p_tensor = VoigtTensor4(self.material_name, 'p', self._params)

        self._anisotropic = True

        # TODO: change to match/case
        if self.crystal == CrystalGroup.Trigonal:
            self.construct_crystal_trigonal()
        elif self.crystal == CrystalGroup.Cubic:
            self.construct_crystal_cubic()
        elif self.crystal == CrystalGroup.GeneralAnisotropic:
            self.construct_crystal_general()

        self.c_tensor.check_symmetries()


    def _add_3d_dispersion_curves_to_axes(self, ax_ivp=None, ax_vg=None):


        axs = []
        if ax_ivp is not None: axs.append(ax_ivp)
        if ax_vg is not None: axs.append(ax_vg)

        # Make data
```

```
        tpts = 50
        ppts = 100
        vphi = np.linspace(0, 2 * np.pi, ppts)
        vtheta = np.linspace(0, np.pi, tpts)

        ivx = np.zeros([tpts, ppts, 3])
        ivy = np.zeros([tpts, ppts, 3])
        ivz = np.zeros([tpts, ppts, 3])

        ivgx = np.zeros([tpts, ppts, 3])
        ivgy = np.zeros([tpts, ppts, 3])
        ivgz = np.zeros([tpts, ppts, 3])

        for ip, phi in enumerate(vphi):
            for itheta, theta in enumerate(vtheta):
                vkap = np.array([np.sin(theta)*np.cos(phi),
                                 np.sin(theta)*np.sin(phi),
                                 np.cos(theta)])
                v_vphase, vecs, v_vgroup = solve_christoffel(vkap, self.c_tensor
, self.rho)

                # slowness curve  eta(vkap) = 1/v_phase(vkap)
                ivx[itheta, ip, :] = vkap[0]/v_vphase
                ivy[itheta, ip, :] = vkap[1]/v_vphase
                ivz[itheta, ip, :] = vkap[2]/v_vphase


                ivgx[itheta, ip, :] = v_vgroup[:, 0]
                ivgy[itheta, ip, :] = v_vgroup[:, 1]
                ivgz[itheta, ip, :] = v_vgroup[:, 2]


        for i in range(3):
            if ax_ivp:
                ax_ivp.plot_surface(ivx[:, :, i], ivy[:, :, i], ivz[:, :, i], al
pha=.25)

            if ax_vg:
                ax_vg.plot_surface(ivgx[:, :, i], ivgy[:, :, i], ivgz[:, :, i],
alpha=.25)


        if ax_ivp:
            ax_ivp.set_xlabel(r'$1/v_x^{(p)}$ [s/km]', fontsize=8, labelpad=1)
            ax_ivp.set_ylabel(r'$1/v_y^{(p)}$ [s/km]', fontsize=8, labelpad=1)
            ax_ivp.set_zlabel(r'$1/v_z^{(p)}$ [s/km]', fontsize=8, labelpad=1)
        if ax_vg:
            ax_vg.set_xlabel(r'$v_x^{(g)}$ [km/s]', fontsize=8, labelpad=1)
            ax_vg.set_ylabel(r'$v_y^{(g)}$ [km/s]', fontsize=8, labelpad=1)
            ax_vg.set_zlabel(r'$v_z^{(g)}$ [km/s]', fontsize=8, labelpad=1)


        for ax in axs:
            for a in ('x', 'y', 'z'):
                ax.tick_params(axis=a, labelsize=8, pad=0)
            for t_ax in [ax.xaxis, ax.yaxis, ax.zaxis]:
                t_ax.line.set_linewidth(.5)

            #ax.set_aspect('equal')

    def plot_bulk_dispersion_3D(self, pref, label=None):
        '''
    Generate isocontour surfaces of the bulk dispersion in 3D k-space.
    '''

        fig, axs = plt.subplots(1,2, subplot_kw={'projection':'3d'})
        ax_vp, ax_vg = axs


        self._add_3d_dispersion_curves_to_axes(ax_vp, ax_vg)


        plt.savefig(pref+'-bulkdisp3D.png')

    def plot_bulk_dispersion(self, pref, label=None):
        '''Draw slowness surface 1/v_p(kappa) and ray surface contours in the horizontal (x-z) plane for the crys
tal axes current orientation.

    Solving the Christoffel equation: D C D^T u = -\rho v_p^2 u, for eigenvalue v_p and eigengector u.
    C is the Voigt form stiffness.
    D = [
    [kapx  0   0   0  kapz  kapy  ]
    [0   kapy  0   kapz 0   kapx  ]
    [0   0   kapz  kapy kapx  0]] where kap=(cos phi, 0, sin phi).

    '''

        fig, axs = setup_bulk_dispersion_2D_plot()

        ax_sl, ax_vp, ax_vg, ax_ivp_3d = axs

        cm = 'cool'  # Color map for polarisation coding
        self._add_bulk_slowness_curves_to_axes(pref, fig, ax_sl, ax_vp, ax_vg, c
m)

        if label is None:
            label = self.material_name
        ax_sl.text(-0.1, 1.1, label, fontsize=14, style='italic', transform=ax_sl.
transAxes)

        self._add_3d_dispersion_curves_to_axes(ax_ivp_3d)

        plt.savefig(pref+'-bulkdisp.png')

    def _add_bulk_slowness_curves_to_axes(self, pref, fig, ax_sl, ax_vp, ax_vg,
cm):

        npolpts = 28
        npolskip = 10  #make bigger
        npts = npolpts*npolskip  # about 1000
        v_kphi = np.linspace(0., np.pi*2, npts)
        v_vel = np.zeros([npts, 3])
        v_velc = np.zeros([npts, 3])
        v_vgx = np.zeros([npts, 3])
        v_vgz = np.zeros([npts, 3])


        cmm = mpl.colormaps[cm]
        with open(pref+'-bulkdisp.dat', 'w') as fout:

            fout.write('#phi   kapx   kapz    vl      vs1      vs2      vlx    vly    vlz    vs1x   vs
1y   vs1z   vs2x   vs2y    vs2z   k.v1   k.v2  k.v3\n')

            kapcomp = np.zeros(3)
```

```
            ycomp = np.zeros(3)
            for ik, kphi in enumerate(v_kphi):
                #kapx = np.cos(kphi)
                #kapz = np.sin(kphi)
                #kapy = 0.0
                vkap = np.array([np.cos(kphi), 0.0, np.sin(kphi)])

                fout.write(f'{kphi:.4f} {vkap[0]:+.4f} {vkap[2]:+.4f} ')


                # solve_christoffel returns:
                # eigvecs are sorted by phase velocity
                # v_vphase[m]:   |vphase| of modes m=1 to 3
                # vecs[:,m]:     evecs of modes m=1 to 3
                # v_vgroup[m,:]  vgroup of mode m, second index is x,y,z
                v_vphase, vecs, v_vgroup = solve_christoffel(vkap, self.c_tensor
, self.rho)

                v_vel[ik, :] = v_vphase    # phase velocity
                v_vgx[ik, :] = v_vgroup[:,0]  # group velocity components
                v_vgz[ik, :] = v_vgroup[:,2]

                ycomp = np.abs(vecs[1,:])                  # $\unity \cdot u_i$
                kapcomp = np.abs(np.matmul(vkap, vecs))  # component of vkap alo
ng each evec
                v_velc[ik, :] = kapcomp    # phase velocity color by polarisatio
n


                for iv in range(3):
                    fout.write(f'{v_vphase[iv]*1000:10.4f} ')
                for iv in range(3):
                    fout.write(f'{vecs[0,iv]:7.4f} {vecs[1,iv]:7.4f}  {vecs[2,iv]:7.4f} ')
                fout.write(f'{kapcomp[0]:6.4f} {kapcomp[1]:6.4f} {kapcomp[2]:6.4f}')

                fout.write('\n')

                # Draw polarisation ball and stick notations
                irad = 0.07/v_vel[0, 0]  # length of polarisation sticks
                rad = 0.07*v_vel[0, 0]  # length of polarisation sticks
                lwstick = .9
                srad = 5  # diameter of polarisation dots
                if ik % npolskip == 0:

                    for i in range(3):
                        radsl = 1/v_vel[ik, i]
                        radvp = v_vel[ik, i]
                        polc = cmm(kapcomp[i])
                        polc = 'k'  # all black for now

                        ptm = radsl*np.array([np.cos(kphi), np.sin(kphi)])
                        pt0 = np.real(ptm - vecs[0:3:2, i]*irad)
                        pt1 = np.real(ptm + vecs[0:3:2, i]*irad)
                        ax_sl.plot((pt0[0], pt1[0]), (pt0[1], pt1[1]), c=polc, l
w=lwstick)
                        ax_sl.plot(ptm[0], ptm[1], 'o', c=polc, markersize=srad*
ycomp[i])

                        ptm = radvp*np.array([np.cos(kphi), np.sin(kphi)])
                        pt0 = np.real(ptm - vecs[0:3:2, i]*rad)
                        pt1 = np.real(ptm + vecs[0:3:2, i]*rad)
                        ax_vp.plot((pt0[0], pt1[0]), (pt0[1], pt1[1]), c=polc, l
w=lwstick)
                        ax_vp.plot(ptm[0], ptm[1], 'o', c=polc, markersize=srad*
ycomp[i])

        # the main curves for 1/v_p and v_g
        for i in range(3):
            ax_sl.scatter(np.cos(v_kphi)/v_vel[:, i], np.sin(v_kphi) /
                       v_vel[:, i], c=v_velc[:, i], vmin=0, vmax=1, s=0.5, cmap=
cm)

            ax_vp.scatter(np.cos(v_kphi)*v_vel[:, i], np.sin(v_kphi) *
                       v_vel[:, i], c=v_velc[:, i], vmin=0, vmax=1, s=0.5, cmap=
cm)

            ax_vg.scatter(v_vgx[:,i], v_vgz[:,i],  c=v_velc[:, i], vmin=0, vmax=
1, s=0.5, cmap=cm)

        # Tick location seems to need help here
        for tax in [ax_vp.xaxis, ax_vp.yaxis, ax_vg.xaxis, ax_vg.yaxis]:
            tax.set_major_locator(ticker.MultipleLocator(2.0, offset=0))

        make_axes_square(np.abs(1/v_vel).max(), ax_sl)
        make_axes_square(np.abs(v_vel).max(), ax_vp)
        make_axes_square(max(np.abs(v_vgx).max(), np.abs(v_vgz).max()), ax_vg)

        #fig.colorbar(mplcm.ScalarMappable(cmap=cm), ax=ax_vp, shrink=.5,
        #              pad=.025, location='top', label='$\hat{e} \cdot \hat{\kapp
a}$')



    def _add_bulk_slowness_curves_to_axes_2x1(self, pref, fig, ax_sl, ax_vp, cm,
 mat1or2):

        npolpts = 28
        npolskip = 10  #make bigger
        npts = npolpts*npolskip  # about 1000
        v_kphi = np.linspace(0., np.pi*2, npts)
        v_vel = np.zeros([npts, 3])
        v_velc = np.zeros([npts, 3])
        v_vgx = np.zeros([npts, 3])
        v_vgz = np.zeros([npts, 3])


        cmm = mpl.colormaps[cm]

        kapcomp = np.zeros(3)
        ycomp = np.zeros(3)
        for ik, kphi in enumerate(v_kphi):
            vkap = np.array([np.cos(kphi), 0.0, np.sin(kphi)])


            # solve_christoffel returns:
            # eigvecs are sorted by phase velocity
            # v_vphase[m]:   |vphase| of modes m=1 to 3
            # vecs[:,m]:     evecs of modes m=1 to 3
            # v_vgroup[m,:]  vgroup of mode m, second index is x,y,z
            v_vphase, vecs, v_vgroup = solve_christoffel(vkap, self.c_tensor, se
lf.rho)

            v_vel[ik, :] = v_vphase    # phase velocity
```

```
            #v_vgx[ik, :] = v_vgroup[:,0]  # group velocity components
            #v_vgz[ik, :] = v_vgroup[:,2]

            ycomp = np.abs(vecs[1,:])                    # $\unity \cdot u_i$
            kapcomp = np.abs(np.matmul(vkap, vecs))  # component of vkap along e
ach evec
            v_velc[ik, :] = kapcomp    # phase velocity color by polarisation



            # Draw polarisation ball and stick notations
            irad = 0.07/v_vel[0, 0]  # length of polarisation sticks
            rad = 0.07*v_vel[0, 0]  # length of polarisation sticks
            lwstick = .9
            srad = 5  # diameter of polarisation dots
            if ik % npolskip == 0:

                for i in range(3):
                    radsl = 1/v_vel[ik, i]
                    radvp = v_vel[ik, i]
                    polc = cmm(kapcomp[i])
                    polc = 'k'  # all black for now

                    ptm = radsl*np.array([np.cos(kphi), np.sin(kphi)])
                    pt0 = np.real(ptm - vecs[0:3:2, i]*irad)
                    pt1 = np.real(ptm + vecs[0:3:2, i]*irad)
                    ax_sl.plot((pt0[0], pt1[0]), (pt0[1], pt1[1]), c=polc, lw=lw
stick)
                    ax_sl.plot(ptm[0], ptm[1], 'o', c=polc, markersize=srad*ycom
p[i])


                    ptm = radvp*np.array([np.cos(kphi), np.sin(kphi)])
                    pt0 = np.real(ptm - vecs[0:3:2, i]*rad)
                    pt1 = np.real(ptm + vecs[0:3:2, i]*rad)

                    #ax_vp.plot((pt0[0], pt1[0]), (pt0[1], pt1[1]), c=polc, lw=l
wstick)
                    #ax_vp.plot(ptm[0], ptm[1], 'o', c=polc, markersize=srad*yco
mp[i])

        # the main curves for 1/v_p and v_g
        for i in range(3):
            ax_sl.scatter(np.cos(v_kphi)/v_vel[:, i], np.sin(v_kphi) /
                       v_vel[:, i], c=v_velc[:, i], vmin=0, vmax=1, s=0.5, cmap
=cm)

            #ax_vp.scatter(np.cos(v_kphi)*v_vel[:, i], np.sin(v_kphi) *
            #             v_vel[:, i], c=v_velc[:, i], vmin=0, vmax=1, s=0.5, cma
p=cm)

            #ax_vg.scatter(v_vgx[:,i], v_vgz[:,i],  c=v_velc[:, i], vmin=0, vmax
=1, s=0.5, cmap=cm)

        # Tick location seems to need help here
        #for tax in [ax_vp.xaxis, ax_vp.yaxis, ax_vg.xaxis, ax_vg.yaxis]:
         #   tax.set_major_locator(ticker.MultipleLocator(2.0, offset=0))

        make_axes_square(np.abs(1/v_vel).max(), ax_sl)
        #make_axes_square(np.abs(v_vel).max(), ax_vp)
        #make_axes_square(max(np.abs(v_vgx).max(), np.abs(v_vgz).max()), ax_vg)

        cbar=fig.colorbar(mplcm.ScalarMappable(cmap=cm), ax=ax_sl, shrink=.5,
```

```python
                        pad=.025, location='right')
        cbar.ax.tick_params(labelsize=6, width=.25)

        cbar.outline.set_linewidth(1)
        cbar.set_label(label=f'Mat {mat1or2} ' +'$\hat{e} \cdot \hat{\kappa}$', fontsize=10)


    def make_crystal_axes_plot(self, pref):
        '''Build crystal coordinates diagram using call to external asymptote application.'''

        fn = tempfile.NamedTemporaryFile(
            suffix='.asy', mode='w+t', delete=False)

        asy_cmds = asy_draw_crystal_axes(self._crystal_axes)
        fn.write(asy_cmds)
        fn.close()

        # run .asy
        subprocess.run(['asy', fn.name, '-o', f'{pref}-crystal'])


def setup_bulk_dispersion_2D_plot():
    '''Plots both slowness and ray normal contours.'''

    fig, axs = plt.subplots(2,2, figsize=(7,6))
    fig.subplots_adjust(hspace=.35, wspace=0)

    ax_sl, ax_vp, ax_vg = axs[0,0], axs[0,1], axs[1,0]

    axs[1,1].set_axis_off()  # Hide axis 2,2

    axs[1,1].remove()
    ax_ivp3d = fig.add_subplot(2,2,4, projection='3d')
    ax_sl.set_xlabel(r'$1/v^{(p)}_{x}$ [s/km]')
    ax_sl.set_ylabel(r'$1/v^{(p)}_{z}$ [s/km]')
    ax_vp.set_xlabel(r'$v^{(p)}_{x}$ [s/km]')
    ax_vp.set_ylabel(r'$v^{(p)}_{z}$ [s/km]')
    ax_vg.set_xlabel(r'$v^{(g)}_{x}$ [km/s]')
    ax_vg.set_ylabel(r'$v^{(g)}_{z}$ [km/s]')

    for ax in axs.flat[:3]:  # Don't write to axis 2,2
        ax.axhline(0, c='gray', lw=.5)
        ax.axvline(0, c='gray', lw=.5)
        ax.tick_params(width=.5)
        for item in ([ax.title, ax.xaxis.label, ax.yaxis.label] +
                     ax.get_xticklabels() + ax.get_yticklabels()):
                item.set_fontsize(10)
        for t_ax in ['top','bottom','left','right']: ax.spines[t_ax].set_linewidth(.5
)
    axs = ax_sl, ax_vp, ax_vg, ax_ivp3d
    return fig, axs


def setup_bulk_dispersion_2D_plot_2x1():
    '''Plots both slowness and ray normal contours.'''

    fig, axs = plt.subplots(1,1, figsize=(6,4))
    #fig.subplots_adjust(hspace=.35, wspace=0)
    axs = axs,
    #ax_sl, ax_vg = axs
    ax_sl = axs[0]
```

```python
    #ax_sl, ax_vp, ax_vg = axs[0,0], axs[0,1], axs[1,0]

    #axs[1,1].set_axis_off()  # Hide axis 2,2

    #axs[1,1].remove()
    #ax_ivp3d = fig.add_subplot(2,2,4, projection='3d')
    ax_sl.set_xlabel(r'$1/v^{(p)}_{x}$ [s/km]')
    ax_sl.set_ylabel(r'$1/v^{(p)}_{z}$ [s/km]')
    #ax_vp.set_xlabel(r'$v^{(p)}_{x}$ [s/km]')
    #ax_vp.set_ylabel(r'$v^{(p)}_{z}$ [s/km]')
    #ax_vg.set_xlabel(r'$v^{(g)}_{x}$ [km/s]')
    #ax_vg.set_ylabel(r'$v^{(g)}_{z}$ [km/s]')

    for ax in axs:  # Don't write to axis 2,2
        ax.axhline(0, c='gray', lw=.5)
        ax.axvline(0, c='gray', lw=.5)
        ax.tick_params(width=.5)
        for item in ([ax.title, ax.xaxis.label, ax.yaxis.label] +
                     ax.get_xticklabels() + ax.get_yticklabels()):
                item.set_fontsize(12)
        for t_ax in ['top','bottom','left','right']: ax.spines[t_ax].set_linewidth(.5
)
    #axs = ax_sl, ax_vp, ax_vg, ax_ivp3d
    return fig, axs


def compare_bulk_dispersion(mat1, mat2, pref):
    fig, axs = setup_bulk_dispersion_2D_plot_2x1()

    #ax_sl, ax_vg = axs
    ax_sl=axs[0]
    ax_vg=None

    cm1 = 'cool'  # Color map for polarisation coding
    cm2 = 'autumn'  # Color map for polarisation coding

    mat1._add_bulk_slowness_curves_to_axes_2x1(pref+'_mat1', fig, ax_sl, ax_vg,
cm1, 1)
    mat2._add_bulk_slowness_curves_to_axes_2x1(pref+'_mat2', fig, ax_sl, ax_vg,
cm2, 2)

    ax_sl.text(0.05, 1.15, f'Mat 1: {mat1.material_name}', fontsize=14, style='italic',
               transform=ax_sl.transAxes)
    ax_sl.text(0.05, 1.05, f'Mat 2: {mat2.material_name}', fontsize=14, style='italic',
               transform=ax_sl.transAxes)

    plt.savefig(pref+'-compare-bulkdisp.png')


def isotropic_stiffness(E, v):
    """
  Calculate the stiffness matrix components of isotropic
  materials, given the two free parameters.

  Ref: www.efunda.com/formulae/solid_mechanics/mat_mechanics/hooke_isotropic.cfm

  Args:
      E  (float): Youngs modulus

      v  (float): Poisson ratio
    """
    c_11 = E*(1-v)/((1+v)*(1-2*v))
    c_12 = E*(v)/((1+v)*(1-2*v))
    c_44 = (E*(1-2*v)/((1+v)*(1-2*v)))/2

    return c_11, c_12, c_44


def asy_draw_crystal_axes(crystal_axes):

    (va, vb, vc) = crystal_axes
    s_avec = '('+','.join(map(str, va))+')'
    s_bvec = '('+','.join(map(str, vb))+')'
    s_cvec = '('+','.join(map(str, vc))+')'

    s1 = '''
settings.outformat='png';
settings.render=8;
import three;
import graph3;

size(2cm,0);
defaultpen(fontsize(7pt));
defaultpen(.2);

real axlen=1.25;
int arrsize=3;
real blen=.5;

//currentprojection=orthographic(1,1,1);
currentprojection=oblique;


draw(O--2X, black, Arrow3(arrsize), L=Label("$\hat{x}$", position=EndPoint));
draw(O--2Y, black, Arrow3(arrsize), L=Label("$\hat{y}$", position=EndPoint));
draw(O--3Z, black, Arrow3(arrsize), L=Label("$\hat{z}$", position=EndPoint));

draw(O-- -2X, gray);
draw(O-- -2Y, gray);
draw(O-- -2Z, gray);


//label("$\hat{x}$", 3X*1.1);
//label("$\hat{y}$", 3Y*1.1);
//label("$\hat{z}$", 3Z*1.1);

draw(box((-1,-.5,-2)*blen,(1,.5,2)*blen),blue);
'''

    s2 = f'''triple avec={s_avec};
triple bvec={s_bvec};
triple cvec={s_cvec};
'''

    s3 = '''triple corig=(0,.5,2)*blen;
draw(corig--avec+corig, red, Arrow3(arrsize), L=Label("$c_x$", position=EndPoint));
draw(corig--bvec+corig, red, Arrow3(arrsize), L=Label("$c_y$", position=EndPoint));
draw(corig--cvec+corig, red, Arrow3(arrsize), L=Label("$c_z$", position=EndPoint));

triple k0=(1,-1,-1);
triple k1=k0+(0,0,2);

draw(k0--k1,green, Arrow3(arrsize), L=Label("$k$"));
'''
```

```

    return s1 + s2 + s3

def make_axes_square(ext0, ax):
    ext = 1.1*ext0
    ax.set_xlim(-ext, ext)
    ax.set_ylim(-ext, ext)
    ax.set_aspect('equal')
```