Source code for pyscf.dft.gen_grid

```
#!/usr/bin/env python
# Copyright 2014-2022 The PySCF Developers. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

'''
Generate DFT grids and weights, based on the code provided by Gerald Knizia <>

Reference for Lebedev-Laikov grid:
  V. I. Lebedev, and D. N. Laikov "A quadrature formula for the sphere of the
  131st algebraic order of accuracy", Doklady Mathematics, 59, 477-481 (1999)
'''


import sys
import ctypes
import numpy
from pyscf import lib
from pyscf.lib import logger
from pyscf.dft import radi
from pyscf import gto
from pyscf.gto.eval_gto import BLKSIZE, NBINS, CUTOFF, make_screen_index
from pyscf import __config__

libdft = lib.load_library('libdft')

GROUP_BOX_SIZE = 1.2
GROUP_BOUNDARY_PENALTY = 4.2
# Padding grids to make the AO value generated by eval_gto aligned in memory
ALIGNMENT_UNIT = 8
NELEC_ERROR_TOL = getattr(__config__, 'dft_rks_prune_error_tol', 0.02)

# ~= (L+1)**2/3
LEBEDEV_ORDER = {
    0  : 1   ,
    3  : 6   ,
    5  : 14  ,
    7  : 26  ,
    9  : 38  ,
    11 : 50  ,
    13 : 74  ,
    15 : 86  ,
    17 : 110 ,
    19 : 146 ,
    21 : 170 ,
    23 : 194 ,
    25 : 230 ,
    27 : 266 ,
    29 : 302 ,
    31 : 350 ,
    35 : 434 ,
    41 : 590 ,
    47 : 770 ,
    53 : 974 ,
    59 : 1202,
    65 : 1454,
    71 : 1730,
    77 : 2030,
    83 : 2354,
    89 : 2702,
    95 : 3074,
    101: 3470,
    107: 3890,
    113: 4334,
    119: 4802,
    125: 5294,
    131: 5810
}
LEBEDEV_NGRID = numpy.array(list(LEBEDEV_ORDER.values()))

# SG0
# S. Chien and P. Gill,  J. Comput. Chem. 27 (2006) 730-739.



[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.sg1_prune]
def sg1_prune(nuc, rads, n_ang, radii=radi.SG1RADII):
    '''SG1, CPL, 209, 506

    Args:
        nuc : int
            Nuclear charge.

        rads : 1D array
            Grid coordinates on radical axis.

        n_ang : int
            Max number of grids over angular part.

    Kwargs:
        radii : 1D array
            radii (in Bohr) for atoms in periodic table

    Returns:
        A list has the same length as rads. The list element is the number of
        grids over angular part for each radial grid.
    '''
# In SG1 the ang grids for the five regions
#            6  38 86  194 86
    leb_ngrid = numpy.array([6, 38, 86, 194, 86])
    alphas = numpy.array((
        (0.25  , 0.5, 1.0, 4.5),
        (0.1667, 0.5, 0.9, 3.5),
        (0.1   , 0.4, 0.8, 2.5)))
    r_atom = radii[nuc] + 1e-200
    if nuc <= 2:  # H, He
        place = ((rads/r_atom).reshape(-1,1) > alphas[0]).sum(axis=1)
    elif nuc <= 10:  # Li - Ne
        place = ((rads/r_atom).reshape(-1,1) > alphas[1]).sum(axis=1)
    else:
        place = ((rads/r_atom).reshape(-1,1) > alphas[2]).sum(axis=1)
    return leb_ngrid[place]
```

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.nwchem_prune]

```
def nwchem_prune(nuc, rads, n_ang, radii=radi.BRAGG_RADII):
    '''NWChem

    Args:
        nuc : int
            Nuclear charge.

        rads : 1D array
            Grid coordinates on radical axis.

        n_ang : int
            Max number of grids over angular part.

    Kwargs:
        radii : 1D array
            radii (in Bohr) for atoms in periodic table

    Returns:
        A list has the same length as rads. The list element is the number of
        grids over angular part for each radial grid.
    '''
    alphas = numpy.array((
        (0.25  , 0.5, 1.0, 4.5),
        (0.1667, 0.5, 0.9, 3.5),
        (0.1   , 0.4, 0.8, 2.5)))
    leb_ngrid = LEBEDEV_NGRID[4:]  # [38, 50, 74, 86, ...]
    if n_ang < 50:
        return numpy.repeat(n_ang, len(rads))
    elif n_ang == 50:
        leb_l = numpy.array([1, 2, 2, 2, 1])
    else:
        idx = numpy.where(leb_ngrid==n_ang)[0][0]
        leb_l = numpy.array([1, 3, idx-1, idx, idx-1])

    r_atom = radii[nuc] + 1e-200
    if nuc <= 2:  # H, He
        place = ((rads/r_atom).reshape(-1,1) > alphas[0]).sum(axis=1)
    elif nuc <= 10:  # Li - Ne
        place = ((rads/r_atom).reshape(-1,1) > alphas[1]).sum(axis=1)
    else:
        place = ((rads/r_atom).reshape(-1,1) > alphas[2]).sum(axis=1)
    angs = leb_l[place]
    angs = leb_ngrid[angs]
    return angs



# Prune scheme JCP 102, 346 (1995); DOI:10.1063/1.469408

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.treutler_prune]
def treutler_prune(nuc, rads, n_ang, radii=None):
    '''Treutler-Ahlrichs

    Args:
        nuc : int
            Nuclear charge.

        rads : 1D array
            Grid coordinates on radical axis.

        n_ang : int
            Max number of grids over angular part.

    Returns:
        A list has the same length as rads. The list element is the number of
        grids over angular part for each radial grid.
    '''
    nr = len(rads)
    leb_ngrid = numpy.empty(nr, dtype=int)
    leb_ngrid[:nr//3] = 14 # l=5
    leb_ngrid[nr//3:nr//2] = 50 # l=11
    leb_ngrid[nr//2:] = n_ang
    return leb_ngrid





###########################################################
# Becke partitioning

# Stratmann, Scuseria, Frisch. CPL, 257, 213 (1996), eq.11

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.stratmann]
def stratmann(g):
    '''Stratmann, Scuseria, Frisch. CPL, 257, 213 (1996); DOI:10.1016/0009-2614(96)00600-
8'''
    a = .64  # for eq. 14
    g = numpy.asarray(g)
    ma = g/a
    ma2 = ma * ma
    g1 = numpy.asarray((1/16.)*(ma*(35 + ma2*(-35 + ma2*(21 - 5 *ma2)))))
    g1[g<=-a] = -1
    g1[g>= a] =  1
    return g1
```

```
[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.original_becke]
def original_becke(g):
    '''Becke, JCP 88, 2547 (1988); DOI:10.1063/1.454033'''
#    This function has been optimized in the C code VXCgen_grid
#    g = (3 - g**2) * g * .5
#    g = (3 - g**2) * g * .5
#    g = (3 - g**2) * g * .5
#    return g
    pass



[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.gen_atomic_grids]
def gen_atomic_grids(mol, atom_grid={}, radi_method=radi.gauss_chebyshev,
                     level=3, prune=nwchem_prune, **kwargs):
    '''Generate number of radial grids and angular grids for the given molecule.

    Returns:
        A dict, with the atom symbol for the dict key.  For each atom type,
        the dict value has two items: one is the meshgrid coordinates wrt the
        atom center; the second is the volume of that grid.
    '''
    if isinstance(atom_grid, (list, tuple)):
        atom_grid = dict([(mol.atom_symbol(ia), atom_grid)
                          for ia in range(mol.natm)])
    atom_grids_tab = {}
    for ia in range(mol.natm):
        symb = mol.atom_symbol(ia)

        if symb not in atom_grids_tab:
            chg = gto.charge(symb)
            if symb in atom_grid:
                n_rad, n_ang = atom_grid[symb]
                if n_ang not in LEBEDEV_NGRID:
                    if n_ang in LEBEDEV_ORDER:
                        logger.warn(mol, 'n_ang %d for atom %d %s is not '
                                    'the supported Lebedev angular grids. '
                                    'Set n_ang to %d', n_ang, ia, symb,
                                    LEBEDEV_ORDER[n_ang])
                        n_ang = LEBEDEV_ORDER[n_ang]
                    else:
                        raise ValueError('Unsupported angular grids %d' % n_ang)
            else:
                n_rad = _default_rad(chg, level)
                n_ang = _default_ang(chg, level)
            rad, dr = radi_method(n_rad, chg, ia, **kwargs)

            rad_weight = 4*numpy.pi * rad**2 * dr

            if callable(prune):
                angs = prune(chg, rad, n_ang)
            else:
                angs = [n_ang] * n_rad
            logger.debug(mol, 'atom %s rad-grids = %d, ang-grids = %s',
                              symb, n_rad, angs)

            angs = numpy.array(angs)
            coords = []
            vol = []
            for n in sorted(set(angs)):
                grid = numpy.empty((n,4))
                libdft.MakeAngularGrid(grid.ctypes.data_as(ctypes.c_void_p),
                                       ctypes.c_int(n))
                idx = numpy.where(angs==n)[0]
                #coords.append(numpy.einsum('i,jk->jik', rad[idx],
grid[:,:3]).reshape(-1,3))
                #vol.append(numpy.einsum('i,j->ji', rad_weight[idx], grid[:,3]).ravel())
                for i0, i1 in lib.prange(0, len(idx), 12):  # 12 radi-grids as a group
                    coords.append(numpy.einsum('i,jk->jik',rad[idx[i0:i1]],
                                               grid[:,:3]).reshape(-1,3))
                    vol.append(numpy.einsum('i,j->ji', rad_weight[idx[i0:i1]],
                                            grid[:,3]).ravel())
            atom_grids_tab[symb] = (numpy.vstack(coords), numpy.hstack(vol))
    return atom_grids_tab
```

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.get_partition]

```
def get_partition(mol, atom_grids_tab,
                  radii_adjust=None, atomic_radii=radi.BRAGG_RADII,
                  becke_scheme=original_becke, concat=True):
    '''Generate the mesh grid coordinates and weights for DFT numerical integration.
    We can change radii_adjust, becke_scheme functions to generate different meshgrid.

    Kwargs:
        concat: bool
            Whether to concatenate grids and weights in return

    Returns:
        grid_coord and grid_weight arrays.  grid_coord array has shape (N,3);
        weight 1D array has N elements.
    '''
    if callable(radii_adjust) and atomic_radii is not None:
        f_radii_adjust = radii_adjust(mol, atomic_radii)
    else:
        f_radii_adjust = None
    atm_coords = numpy.asarray(mol.atom_coords() , order='C')
    atm_dist = gto.inter_distance(mol)
    if (becke_scheme is original_becke and
        (radii_adjust is radi.treutler_atomic_radii_adjust or
         radii_adjust is radi.becke_atomic_radii_adjust or
         f_radii_adjust is None)):
        if f_radii_adjust is None:
            p_radii_table = lib.c_null_ptr()
        else:
            f_radii_table = numpy.asarray([f_radii_adjust(i, j, 0)
                                           for i in range(mol.natm)
                                           for j in range(mol.natm)])
            p_radii_table = f_radii_table.ctypes.data_as(ctypes.c_void_p)

        def gen_grid_partition(coords):
            coords = numpy.asarray(coords, order='F')
            ngrids = coords.shape[0]
            pbecke = numpy.empty((mol.natm,ngrids))
            libdft.VXCgen_grid(pbecke.ctypes.data_as(ctypes.c_void_p),
                               coords.ctypes.data_as(ctypes.c_void_p),
                               atm_coords.ctypes.data_as(ctypes.c_void_p),
                               p_radii_table,
                               ctypes.c_int(mol.natm), ctypes.c_int(ngrids))
            return pbecke
    else:
        def gen_grid_partition(coords):
            ngrids = coords.shape[0]
            grid_dist = numpy.empty((mol.natm,ngrids))
            for ia in range(mol.natm):
                dc = coords - atm_coords[ia]
                grid_dist[ia] = numpy.sqrt(numpy.einsum('ij,ij->i',dc,dc))
            pbecke = numpy.ones((mol.natm,ngrids))
            for i in range(mol.natm):
                for j in range(i):
                    g = 1/atm_dist[i,j] * (grid_dist[i]-grid_dist[j])
                    if f_radii_adjust is not None:
                        g = f_radii_adjust(i, j, g)
                    g = becke_scheme(g)
                    pbecke[i] *= .5 * (1-g)
                    pbecke[j] *= .5 * (1+g)
            return pbecke

    coords_all = []
    weights_all = []
    for ia in range(mol.natm):
        coords, vol = atom_grids_tab[mol.atom_symbol(ia)]
        coords = coords + atm_coords[ia]
        pbecke = gen_grid_partition(coords)
        weights = vol * pbecke[ia] * (1./pbecke.sum(axis=0))
        coords_all.append(coords)
        weights_all.append(weights)

    if concat:
        coords_all = numpy.vstack(coords_all)
        weights_all = numpy.hstack(weights_all)
    return coords_all, weights_all


gen_partition = get_partition


[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.make_mask]
def make_mask(mol, coords, relativity=0, shls_slice=None, cutoff=CUTOFF,
              verbose=None):
    '''Mask to indicate whether a shell is ignorable on grids. See also the
    function gto.eval_gto.make_screen_index

    Args:
        mol : an instance of :class:`Mole`

        coords : 2D array, shape (N,3)
            The coordinates of grids.

    Kwargs:
        relativity : bool
            No effects.
        shls_slice : 2-element list
            (shl_start, shl_end).
            If given, only part of AOs (shl_start <= shell_id < shl_end) are
            evaluated.  By default, all shells defined in mol will be evaluated.
        verbose : int or object of :class:`Logger`
            No effects.

    Returns:
        2D mask array of shape (N,nbas), where N is the number of grids, nbas
        is the number of shells.
    '''
    return make_screen_index(mol, coords, shls_slice, cutoff)



[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.arg_group_grids]
def arg_group_grids(mol, coords, box_size=GROUP_BOX_SIZE):
    '''
    Partition the entire space into small boxes according to the input box_size.
    Group the grids against these boxes.
    '''
    atom_coords = mol.atom_coords()
    boundary = [atom_coords.min(axis=0) - GROUP_BOUNDARY_PENALTY,
                atom_coords.max(axis=0) + GROUP_BOUNDARY_PENALTY]
    # how many boxes inside the boundary
    boxes = ((boundary[1] - boundary[0]) * (1./box_size)).round().astype(int)
    tot_boxes = numpy.prod(boxes + 2)
    logger.debug(mol, 'tot_boxes %d, boxes in each direction %s', tot_boxes, boxes)
    # box_size is the length of each edge of the box
    box_size = (boundary[1] - boundary[0]) / boxes
    frac_coords = (coords - boundary[0]) * (1./box_size)
    box_ids = numpy.floor(frac_coords).astype(int)
    box_ids[box_ids<-1] = -1
    box_ids[box_ids[:,0] > boxes[0], 0] = boxes[0]
    box_ids[box_ids[:,1] > boxes[1], 1] = boxes[1]
    box_ids[box_ids[:,2] > boxes[2], 2] = boxes[2]
    rev_idx, counts = numpy.unique(box_ids, axis=0, return_inverse=True,
                                   return_counts=True)[1:3]
    return rev_idx.argsort(kind='stable')



def _load_conf(mod, name, default):
    var = getattr(__config__, name, None)
    if var is None:
        var = default
    elif isinstance(var):
        if mod is None:
            mod = sys.modules[__name__]
        var = getattr(mod, var)

    if callable(var):
        return staticmethod(var)
    else:
        return var
```

```
[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids]
class Grids(lib.StreamObject):
    '''DFT mesh grids

    Attributes for Grids:
        level : int
            To control the number of radial and angular grids. Large number
            leads to large mesh grids. The default level 3 corresponds to
            (50,302) for H, He;
            (75,302) for second row;
            (80~105,434) for rest.

            Grids settings at other levels can be found in
            pyscf.dft.gen_grid.RAD_GRIDS and pyscf.dft.gen_grid.ANG_ORDER

        atomic_radii : 1D array
            | radi.BRAGG_RADII  (default)
            | radi.COVALENT_RADII
            | None : to switch off atomic radii adjustment

        radii_adjust : function(mol, atomic_radii) => (function(atom_id, atom_id, g) =>
array_like_g)
            Function to adjust atomic radii, can be one of
            | radi.treutler_atomic_radii_adjust
            | radi.becke_atomic_radii_adjust
            | None : to switch off atomic radii adjustment

        radi_method : function(n) => (rad_grids, rad_weights)
            scheme for radial grids, can be one of
            | radi.treutler  (default)
            | radi.delley
            | radi.mura_knowles
            | radi.gauss_chebyshev

        becke_scheme : function(v) => array_like_v
            weight partition function, can be one of
            | gen_grid.original_becke  (default)
            | gen_grid.stratmann

        prune : function(nuc, rad_grids, n_ang) => list_n_ang_for_each_rad_grid
            scheme to reduce number of grids, can be one of
            | gen_grid.nwchem_prune  (default)
            | gen_grid.sg1_prune
            | gen_grid.treutler_prune
            | None : to switch off grid pruning

        symmetry : bool
            whether to symmetrize mesh grids (TODO)

        atom_grid : dict
            Set (radial, angular) grids for particular atoms.
            Eg, grids.atom_grid = {'H': (20,110)} will generate 20 radial
            grids and 110 angular grids for H atom.

        Examples:

        >>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
        >>> grids = dft.gen_grid.Grids(mol)
        >>> grids.level = 4
        >>> grids.build()
        '''

    atomic_radii = _load_conf(radi, 'dft_gen_grid_Grids_atomic_radii',
                               radi.BRAGG_RADII)
    radii_adjust = _load_conf(radi, 'dft_gen_grid_Grids_radii_adjust',
                               radi.treutler_atomic_radii_adjust)
    radi_method = _load_conf(radi, 'dft_gen_grid_Grids_radi_method',
                              radi.treutler)
    becke_scheme = _load_conf(None, 'dft_gen_grid_Grids_becke_scheme',
                               original_becke)
    prune = _load_conf(None, 'dft_gen_grid_Grids_prune', nwchem_prune)
    level = getattr(__config__, 'dft_gen_grid_Grids_level', 3)

    alignment = ALIGNMENT_UNIT
    cutoff = CUTOFF

    _keys = set((
        'atomic_radii', 'radii_adjust', 'radi_method', 'becke_scheme',
        'prune', 'level', 'alignment', 'cutoff', 'mol', 'symmetry',
        'atom_grid', 'non0tab', 'screen_index', 'coords', 'weights',
    ))

    def __init__(self, mol):
        self.mol = mol
        self.stdout = mol.stdout
        self.verbose = mol.verbose
        self.symmetry = mol.symmetry
        self.atom_grid = {}

##################################################
# don't modify the following attributes, they are not input options
        self.non0tab = None
        # Integral screen index ~= NBINS + log(ao).
        # screen_index > 0 for non-zero AOs
        self.screen_index = None
        self.coords  = None
        self.weights = None

    @property
    def size(self):
        return getattr(self.weights, 'size', 0)

    def __setattr__(self, key, val):
        if key in ('atom_grid', 'atomic_radii', 'radii_adjust', 'radi_method',
                   'becke_scheme', 'prune', 'level'):
            self.reset()
        super(Grids, self).__setattr__(key, val)


[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids.dump_flags]
    def dump_flags(self, verbose=None):
        logger.info(self, 'radial grids: %s', self.radi_method.__doc__)
        logger.info(self, 'becke partition: %s', self.becke_scheme.__doc__)
        logger.info(self, 'pruning grids: %s', self.prune)
        logger.info(self, 'grids dens level: %d', self.level)
        logger.info(self, 'symmetrized grids: %s', self.symmetry)
        if self.radii_adjust is not None:
            logger.info(self, 'atomic radii adjust function: %s',
                        self.radii_adjust)
            logger.debug2(self, 'atomic_radii : %s', self.atomic_radii)
        if self.atom_grid:
            logger.info(self, 'User specified grid scheme %s', str(self.atom_grid))
        return self
```

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids.build]

```
    def build(self, mol=None, with_non0tab=False, sort_grids=True, **kwargs):
        if mol is None: mol = self.mol
        if self.verbose >= logger.WARN:
            self.check_sanity()
        atom_grids_tab = self.gen_atomic_grids(
            mol, self.atom_grid, self.radi_method, self.level, self.prune, **kwargs)
        self.coords, self.weights = self.get_partition(
            mol, atom_grids_tab, self.radii_adjust, self.atomic_radii, self.becke_scheme)

        if sort_grids:
            idx = arg_group_grids(mol, self.coords)
            self.coords = self.coords[idx]
            self.weights = self.weights[idx]

        if self.alignment > 1:
            padding = _padding_size(self.size, self.alignment)
            logger.debug(self, 'Padding %d grids', padding)
            if padding > 0:
                self.coords = numpy.vstack(
                    [self.coords, numpy.repeat([[1e-4]*3], padding, axis=0)])
                self.weights = numpy.hstack([self.weights, numpy.zeros(padding)])

        if with_non0tab:
            self.non0tab = self.make_mask(mol, self.coords)
            self.screen_index = self.non0tab
        else:
            self.screen_index = self.non0tab = None
        logger.info(self, 'tot grids = %d', len(self.weights))
        return self
```

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids.kernel]

```
    def kernel(self, mol=None, with_non0tab=False):
        self.dump_flags()
        return self.build(mol, with_non0tab=with_non0tab)
```

[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids.reset]

```
    def reset(self, mol=None):
        '''Reset mol and clean up relevant attributes for scanner mode'''
        if mol is not None:
            self.mol = mol
        self.coords = None
        self.weights = None
        self.non0tab = None
        self.screen_index = None
        return self



    gen_atomic_grids = lib.module_method(
        gen_atomic_grids, ['atom_grid', 'radi_method', 'level', 'prune'])


[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids.get_partition]
    @lib.with_doc(get_partition.__doc__)
    def get_partition(self, mol, atom_grids_tab=None,
                      radii_adjust=None, atomic_radii=radi.BRAGG_RADII,
                      becke_scheme=original_becke, concat=True):
        if atom_grids_tab is None:
            atom_grids_tab = self.gen_atomic_grids(mol)
        return get_partition(mol, atom_grids_tab, radii_adjust, atomic_radii,
                             becke_scheme, concat=concat)



    gen_partition = get_partition

    make_mask = lib.module_method(make_mask, absences=['cutoff'])


[docs] [../../../pyscf_api_docs/pyscf.dft.html#pyscf.dft.gen_grid.Grids.prune_by_density_]
    def prune_by_density_(self, rho, threshold=0):
        '''Prune grids if the electron density on the grid is small'''
        if threshold == 0:
            return self

        mol = self.mol
        n = numpy.dot(rho, self.weights)
        if abs(n-mol.nelectron) < NELEC_ERROR_TOL*n:
            rho *= self.weights
            idx = abs(rho) > threshold / self.weights.size
            logger.debug(self, 'Drop grids %d',
                         self.weights.size - numpy.count_nonzero(idx))
            self.coords  = numpy.asarray(self.coords [idx], order='C')
            self.weights = numpy.asarray(self.weights[idx], order='C')
            if self.alignment > 1:
                padding = _padding_size(self.size, self.alignment)
                logger.debug(self, 'prune_by_density_: %d padding grids', padding)
                if padding > 0:
                    self.coords = numpy.vstack(
                        [self.coords, numpy.repeat([[1e-4]*3], padding, axis=0)])
                    self.weights = numpy.hstack([self.weights, numpy.zeros(padding)])
            self.non0tab = self.make_mask(mol, self.coords)
            self.screen_index = self.non0tab
        return self




def _default_rad(nuc, level=3):
    '''Number of radial grids '''
    tab   = numpy.array( (2 , 10, 18, 36, 54, 86, 118))
    period = (nuc > tab).sum()
    return RAD_GRIDS[level,period]
#                Period    1   2   3   4   5   6   7         # level
RAD_GRIDS = numpy.array((( 10, 15, 20, 30, 35, 40, 50),      # 0
                         ( 30, 40, 50, 60, 65, 70, 75),      # 1
                         ( 40, 60, 65, 75, 80, 85, 90),      # 2
                         ( 50, 75, 80, 90, 95,100,105),      # 3
                         ( 60, 90, 95,105,110,115,120),      # 4
                         ( 70,105,110,120,125,130,135),      # 5
                         ( 80,120,125,135,140,145,150),      # 6
                         ( 90,135,140,150,155,160,165),      # 7
                         (100,150,155,165,170,175,180),      # 8
                         (200,200,200,200,200,200,200),))    # 9

def _default_ang(nuc, level=3):
    '''Order of angular grids. See LEBEDEV_ORDER for the mapping of
    the order and the number of angular grids'''
    tab   = numpy.array( (2 , 10, 18, 36, 54, 86, 118))
    period = (nuc > tab).sum()
    return LEBEDEV_ORDER[ANG_ORDER[level,period]]
#               Period    1   2   3   4   5   6   7          # level
ANG_ORDER = numpy.array(((11, 15, 17, 17, 17, 17, 17 ),      # 0
                         (17, 23, 23, 23, 23, 23, 23 ),      # 1
                         (23, 29, 29, 29, 29, 29, 29 ),      # 2
                         (29, 29, 35, 35, 35, 35, 35 ),      # 3
                         (35, 41, 41, 41, 41, 41, 41 ),      # 4
                         (41, 47, 47, 47, 47, 47, 47 ),      # 5
                         (47, 53, 53, 53, 53, 53, 53 ),      # 6
                         (53, 59, 59, 59, 59, 59, 59 ),      # 7
                         (59, 59, 59, 59, 59, 59, 59 ),      # 8
                         (65, 65, 65, 65, 65, 65, 65 ),))    # 9

def _padding_size(ngrids, alignment):
    if alignment <= 1:
        return 0
    return (ngrids + alignment - 1) // alignment * alignment - ngrids
```