


Crystal structure

Modules: [pyscf.pbc.gto](#) [[../pyscf_api_docs/pyscf.pbc.gto.html#module-pyscf.pbc.gto](#)]

Initializing a crystal

Initializing a crystal unit cell is very similar to the initialization of a molecule. Instead of the `gto.Mole` class, one uses the `gto.Cell` class to define a cell:



```
>>> from pyscf.pbc import gto
>>> cell = gto.Cell()
>>> cell.atom = '''H 0 0 0; H 1 1 1'''
>>> cell.basis = 'gth-dzvp'
>>> cell.pseudo = 'gth-pade'
>>> cell.a = numpy.eye(3) * 2
>>> cell.build()
```

The other two equivalent ways to initialize a molecule introduced in [Molecular structure](#) [[../gto.html#user-gto](#)] also apply here:

```
>>> from pyscf.pbc import gto
>>> cell = gto.Cell()
>>> cell.build(
...     atom = '''H 0 0 0; H 1 1 1''',
...     basis = 'gth-dzvp',
...     pseudo = 'gth-pade',
...     a = numpy.eye(3) * 2)

>>> import pyscf
>>> cell = pyscf.M(
...     atom = '''H 0 0 0; H 1 1 1''',
...     basis = 'gth-dzvp',
```

```
... pseudo = 'gth-pade',
... a = numpy.eye(3) * 2)

>>> from pyscf.pbc import gto
>>> cell = gto.M(
...     atom = '''H 0 0 0; H 1 1 1''',
...     basis = 'gth-dzvp',
...     pseudo = 'gth-pade',
...     a = numpy.eye(3) * 2)
```

Lattice vectors

The crystal initialization requires an additional parameter **a**, which represents the lattice vectors. The format of **a** is array-like:

```
cell.a = numpy.eye(3) * 2
cell.a = [[2,0,0],[0,2,0],[0,0,2]]
```

Each row of the 3-by-3 matrix of **a** represents a lattice vector in Cartesian coordinate, with the same unit as the input **atom** parameter.

Note

The input lattice vectors (row by row) should form a right-handed coordinate system, as otherwise some integrals may be computed incorrectly in PySCF.

Basis set and pseudopotential

PySCF uses crystalline Gaussian-type orbitals as basis functions for solid calculations. The predefined basis sets and ECPs for molecular calculations can be used in solid calculations as well. In addition, the predefined basis sets include valence basis sets that are optimized for the GTH pseudopotentials (a whole list can be found in [pyscf/pbc/gto/basis](https://github.com/pyscf/pyscf/tree/master/pyscf/pbc/gto/basis) [https://github.com/pyscf/pyscf/tree/master/pyscf/pbc/gto/basis] and [pyscf/pbc/gto/pseudo](https://github.com/pyscf/pyscf/tree/master/pyscf/pbc/gto/pseudo)

[<https://github.com/pyscf/pyscf/tree/master/pyscf/pbc/gto/pseudo>]). The input format of [basis sets](#) [[./gto.html#basis-sets](#)] for the `cell` object is the same as that for the `Mole` object.

```
#!/usr/bin/env python

'''
Input pseudo potential using functions pbc.gto.pseudo.parse and
pbc.gto.pseudo.load

It is allowed to mix the Quantum chemistry effective core potential
(ECP) with
crystal pseudo potential (PP). Input ECP with .ecp attribute and PP
with
.pseudo attribute.

See also
pyscf/pbc/gto/pseudo/GTH_POTENTIALS for the GTH-potential format
pyscf/examples/gto/05-input_ecp.py for quantum chemistry ECP format
'''

import numpy
from pyscf.pbc import gto

cell = gto.M(atom='''
Si1 0 0 0
Si2 1 1 1''',
              a = '''3    0    0
                   0    3    0
                   0    0    3''',
              basis = {'Si1': 'gth-szv', # Goedecker, Teter and Hutter
                       'Si2': 'lanl2dz'},
              pseudo = {'Si1': gto.pseudo.parse('''
Si
  2    2
  0.44000000    1    -6.25958674
  2
  0.44465247    2    8.31460936    -2.33277947
                                3.01160535
  0.50279207    1    2.33241791
''')},
              ecp = {'Si2': 'lanl2dz'}, # ECP for second Si atom
              )
```

```

#
# Some elements have multiple PP definitions in GTH database. Add
# suffix in
# the basis name to load the specific PP.
#
cell = gto.M(
    a = numpy.eye(3)*5,
    atom = 'Mg1 0 0 0; Mg2 0 0 1',
    pseudo = {'Mg1': 'gth-lda-q2', 'Mg2': 'gth-lda-q10'})

#
# Allow mixing quantum chemistry ECP (or BFD PP) and crystal PP in the
# same calculation.
#
cell = gto.M(
    a = '''4      0      0
           0      4      0
           0      0      4''',
    atom = 'Cl 0 0 1; Na 0 1 0',
    basis = {'na': 'gth-szv', 'Cl': 'bfd-vdz'},
    ecp = {'Cl': 'bfd-pp'},
    pseudo = {'Na': 'gthbp'})

#
# ECP can be input in the attribute .pseudo
#
cell = gto.M(
    a = '''4      0      0
           0      4      0
           0      0      4''',
    atom = 'Cl 0 0 1; Na 0 1 0',
    basis = {'na': 'gth-szv', 'Cl': 'bfd-vdz'},
    pseudo = {'Na': 'gthbp', 'Cl': 'bfd-pp'})

```

Finally, custom basis sets can be defined just like in molecular calculations

```

#!/usr/bin/env python

'''
Basis can be input the same way as the finite-size system.
'''

```

```

#
# Note pbc.gto.parse does not support NWChem format. To parse NWChem
# format
# basis string, you need the molecule gto.parse function.
#

import numpy
from pyscf import gto
from pyscf.pbc import gto as pgto
cell = pgto.M(
    atom = '''C      0.      0.      0.
              C      0.8917  0.8917  0.8917
              C      1.7834  1.7834  0.
              C      2.6751  2.6751  0.8917
              C      1.7834  0.      1.7834
              C      2.6751  0.8917  2.6751
              C      0.      1.7834  1.7834
              C      0.8917  2.6751  2.6751''',
    basis = {'C': gto.parse(''
# Parse NWChem format basis string (see
https://bse.pnl.gov/bse/portal).
# Comment lines are ignored
#BASIS SET: (6s,3p) -> [2s,1p]
0    S
    130.7093200          0.15432897
    23.8088610          0.53532814
    6.4436083           0.44463454
0    SP
    5.0331513          -0.09996723          0.15591627
    1.1695961          0.39951283          0.60768372
    0.3803890          0.70011547          0.39195739
    ''')},
    pseudo = 'gth-pade',
    a = numpy.eye(3)*3.5668)

```

Low-dimensional systems

The PySCF `pbc` module also supports low-dimensional periodic systems. You can initialize the attribute `Cell.dimension` to specify the dimension of the system:

```

>>> cell.dimension = 2
>>> cell.a = numpy.eye(3) * 2

```

```
>>> cell.build()
```

When **dimension** is smaller than 3, a vacuum of infinite size will be applied in certain direction(s). For example, when **dimension** is 2, the z-direction will be treated as infinitely large and the xy-plane constitutes the periodic surface. When **dimension** is 1, the y and z axes are treated as vacuum and thus the system is a wire in the x direction. When **dimension** is 0, all three directions are treated as vacuum, and this is equivalent to a molecular system.

K-points

The k-points used in solid calculations can be obtained through the `Cell.make_kpts()` method. The minimal input is to specify the number of k-points in each lattice vector direction:

```
>>> kpts = cell.make_kpts([1,2,2])
>>> print(kpts.shape)
(4,3)
```

By default, this will return the shifted Monkhorst-Pack mesh which is centered at the *Gamma*-point. To get the non-shifted Monkhorst-Pack mesh, one can call:

```
>>> kpts = cell.make_kpts([1,2,2], with_gamma_point=False)
```

To get a shifted Monkhorst-pack mesh centered at a specific point, one can call:

```
>>> kpts = cell.make_kpts([1,2,2], scaled_center=[0.,0.25,0.25])
```

where `scaled_center` is defined in the units of lattice vectors.

The obtained k-points are used as input for crystalline electronic structure calculations. For example, k-point sampled RHF can be invoked as follows:

```
>>> from pyscf.pbc import scf
>>> kpts = cell.make_kpts([2,2,2])
```

```
>>> kmf = scf.KRHF(cell, kpts = kpts)
>>> kmf.kernel()
```

More details about k-point sampling for each method can be found in the corresponding chapters.

Spin

The attribute `spin` of `Cell` class has a different meaning to the `spin` of `Mole` class. `Mole.spin` indicates the number of unpaired electrons of a molecule. Most attributes of class `Cell` represents the information of a unit cell, `Cell.spin` is used to indicate the number of unpaired electrons of a super cell. In a Γ -point calculation, one can set `cell.spin` to control the unpaired electrons of the entire system. For calculations with k-point samplings, for example a 2x2x2 k-point calculation, `cell.spin=1` leads to one unpaired electron rather than 8 unpaired electrons.

If wrong `cell.spin` is set, calculations will still run and a warning message for inconsistency between the electron number and spin may be raised.

Currently, the program does not support to assign the unpaired electrons per unit cell. A temporary workaround is to set `cell.spin` to the product of the number of unpaired electrons and k-points. However, this setting only guarantees that the total numbers of alpha electrons and beta electrons agree to the `cell.spin` cross all k-points. The occupancies may be different at different k-points.

Other parameters

The attribute `precision` of `Cell` object determines the integral accuracy. The default value is `1e-8` hartree. When calling the `cell.build()` method, some parameters are set automatically based on the value of `precision`, including:

- **mesh** – length-3 list or 1x3 array of int
 - The numbers of grid points in the FFT-mesh in each direction.
- **ke_cutoff** – float

- Kinetic energy cutoff of the plane waves in FFT-DF
- **rcut** – float
 - Cutoff radius (in Bohr) of the lattice summation in the integral evaluation

Other attributes of the **Mole** class such as **verbose**, **max_memory**, etc., have the same meanings in the **Cell** class.

Note

Currently, point group symmetries for crystals are not supported.

Accessing AO integrals

The **Mole.intor()** method can only be used to evaluate integrals with open boundary conditions. When the periodic boundary conditions of crystalline systems are used, one needs to use the **pbccell.pbccell.intor()** function to evaluate the integrals for short-range operators, such as the overlap and kinetic matrix:

```
overlap = cell.pbccell.intor('int1e_ovlp')
kin = cell.pbccell.intor('int1e_kin')
```

By default, the **pbccell.pbccell.intor()** function only returns the Γ -point integrals. If k-points are specified, it will evaluate the integrals at each k-point:

```
kpts = cell.make_kpts([2,2,2]) # 2x2x2 Monkhorst-pack mesh
overlap = cell.pbccell.intor('int1e_ovlp', kpts=kpts)
```

Note

pbccell.pbccell.intor() can only be used to evaluate the short-range integrals. PBC density fitting methods have to be used to compute the integrals for long-range operators such as nuclear attraction and Coulomb repulsion.

The two-electron Coulomb integrals can be evaluated with the PBC density fitting methods:

```
from pyscf.pbc import df
eri = df.DF(cell).get_eri()
```

See [Density fitting for crystalline calculations](#) [df.html#user-pbc-df] for more details of the PBC density fitting methods.