Molecular structure

Modules: **pyscf.gto** [../pyscf_api_docs/pyscf.gto.html#module-pyscf.gto]

Initializing a molecule

There are three ways to define and initialize a molecule. The first is to use the keyword arguments of the `Mole.build()` method to initialize a molecule:

✓
```
>>> from pyscf import gto
>>> mol = gto.Mole()
>>> mol.build(
...     atom = '''O 0 0 0; H  0 1 0; H 0 0 1''',
...     basis = 'sto-3g')
```

The second way is to assign the geometry, basis etc., to the `Mole` object, followed by calling the `build()` method:

✓
```
>>> from pyscf import gto
>>> mol = gto.Mole()
>>> mol.atom = '''O 0 0 0; H  0 1 0; H 0 0 1'''
>>> mol.basis = 'sto-3g'
>>> mol.build()
```

The third way is to use the shortcut functions `pyscf.M()` [../pyscf_api_docs/pyscf.html#pyscf.M] or `Mole.M()`. These functions pass all the arguments to the `build()` method:

✓
```
>>> import pyscf
>>> mol = pyscf.M(
...     atom = '''O 0 0 0; H  0 1 0; H 0 0 1''',
...     basis = 'sto-3g')

>>> from pyscf import gto
>>> mol = gto.M(
...     atom = '''O 0 0 0; H  0 1 0; H 0 0 1''',
...     basis = 'sto-3g')
```

In any of these, you may have noticed two keywords `atom` and `basis`. They are used to hold the molecular geometry [#geometry] and basis sets [#basis-sets], which can be defined along with other input options as follows.

Geometry

The molecular geometry can be input in Cartesian format with the default unit being Angstrom (one can specify the unit by setting the attribute **`unit`** to either `'Angstrom'` or `'Bohr'`):

✓

```
mol = gto.Mole()
mol.atom = '''
    O   0. 0. 0.
    H   0. 1. 0.
    H   0. 0. 1.
'''
mol.unit = 'B' # case insensitive, any string not starts by 'B' or
'AU' is treated as 'Angstrom'
```

The atoms in the molecule are represented by an element symbol plus three numbers for coordinates. Different atoms should be separated by `;` or a line break. In the same atom, `,` can be used to separate different items. Blank lines or lines started with `#` will be ignored:

```
>>> mol = pyscf.M(
... atom = '''
... #O 0 0 0
... H 0 1 0
...
... H 0 0 1
... ''')
```

```
>>> mol.natm
2
```

The input parser also supports the Z-matrix input format:

```
mol = gto.Mole()
mol.atom = '''
    O
    H  1  1.2
    H  1  1.2  2 105
'''
```

Similarly, different atoms need to be separated by `;` or a line break.

The geometry string is case-insensitive. It also supports to input the nuclear charges of elements:

```
>>> mol = gto.Mole()
>>> mol.atom = '''8 0. 0. 0.; h 0. 1. 0; H 0. 0. 1.'''
```

If you need to label an atom to distinguish it from the others, you can prefix or suffix the atom symbol with a number `1234567890` or a special character `~!@#$%^&*()_+.?:<>[]{}|` (not `,` and `;`). With this decoration, you can specify different basis sets, masses, or nuclear models on different atoms:

```
>>> mol = gto.Mole()
>>> mol.atom = '''8 0 0 0; h:1 0 1 0; H@2 0 0 1'''
>>> mol.unit = 'B'
>>> mol.basis = {'O': 'sto-3g', 'H': 'cc-pvdz', 'H@2': '6-31G'}
>>> mol.build()
>>> print(mol._atom)
[['O', [0.0, 0.0, 0.0]], ['H:1', [0.0, 1.0, 0.0]], ['H@2', [0.0, 0.0,
1.0]]]
```

You can also input the geometry in the internal format of **`Mole.atom`**:

```
atom = [[atom1, (x, y, z)],
        [atom2, (x, y, z)],
        ...
        [atomN, (x, y, z)]]
```

This is convenient as you can use Python script to construct the geometry:

```
>>> mol = gto.Mole()
>>> mol.atom = [['O',(0, 0, 0)], ['H',(0, 1, 0)], ['H',(0, 0, 1)]]
>>> mol.atom.extend([['H', (i, i, i)] for i in range(1,5)])
```

Besides Python list, tuple and numpy.ndarray are all supported by the internal format:

```
>>> mol.atom = (('O',numpy.zeros(3)), ['H', 0, 1, 0], ['H',[0, 0, 1]])
```

You can also specify the path to an xyz file and PySCF will use the coordinates from this file to build **`Mole.atom`**.

```
✓>>> mol = gto.M(atom="my_molecule.xyz")
```

Or:

```
>>> mol = gto.Mole()
✓>>> mol.atom = "my_molecule.xyz"
>>> mol.build()
```

No matter which format or symbols are used in the input, **`Mole.build()`** will convert **`Mole.atom`** to the internal format:

```
>>> mol.atom = '''
    O        0,    0, 0             ; 1 0.0 1 0

        H@2,0 0 1
    '''
>>> mol.unit = 'B'
>>> mol.build()
>>> print(mol._atom)
[('O', [0.0, 0.0, 0.0]), ('H', [0.0, 1.0, 0.0]), ('H@2', [0.0, 0.0,
1.0])]
```

which is stored as the attribute **`Mole._atom`**.

Once the **`Mole`** object is built, the molecular geometry can be accessed through the **`Mole.atom_coords()`** function. This function returns a (N,3) array for the coordinates of each atom:

```
>>> print(mol.atom_coords(unit='Bohr')) # unit can be "ANG" or "Bohr"
[[0. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

Ghost atoms can also be specified when inputting the geometry. See examples/gto/03-ghost_atom.py [https://github.com/pyscf/pyscf/tree/master/examples/gto/03-ghost_atom.py] for examples.

Basis set

The simplest way to define the basis set is to assign the name of the basis as a string to **`Mole.basis`**:

```
✓mol.basis = 'sto-3g'
```

This input will apply the specified basis set to all atoms. The name of the basis set in the string is case insensitive. White spaces, dashes and underscores in the name are all ignored. If different basis sets are required for different elements, a Python `dict` can be used:

```
✓mol.basis = {'O': 'sto-3g', 'H': '6-31g'}
```

One can also input custom basis sets with the helper functions. The function **`gto.basis.parse()`** can parse a basis string in the NWChem format (https://bse.pnl.gov/bse/portal [https://bse.pnl.gov/bse/portal]):

```
mol.basis = {'O': gto.basis.parse('''
C    S
```

```
     71.6168370              0.15432897
     13.0450960              0.53532814
      3.5305122              0.44463454
C    SP
      2.9412494             -0.09996723             0.15591627
      0.6834831              0.39951283             0.60768372
      0.2222899              0.70011547             0.39195739
''')}
```

The functions **`gto.basis.load()`** can load arbitrary basis sets from the database, even if the basis set does not match the element:

```
mol.basis = {'H': gto.basis.load('sto3g', 'C')}
```

Both **`gto.basis.parse()`** and **`gto.basis.load()`** return the basis set in the internal format (see Basis format [#basis-format]).

The basis parser also supports ghost atoms:

```
mol.basis = {'GHOST': gto.basis.load('cc-pvdz', 'O'), 'H': 'sto3g'}
```

More examples of ghost atoms can be found in examples/gto/03-ghost_atom.py [https://github.com/pyscf/pyscf/tree/master/examples/gto/03-ghost_atom.py].

Like the requirements for geometry input, you can use atomic symbols (case-insensitive) or atomic nuclear charges as the keyword of the **`basis`** dictionary. Prefix and suffix of numbers and special characters are allowed. If the decorated atomic symbol is appeared in **`atom`** but not in **`basis`**, the basis parser will remove all decorations and seek the pure atomic symbol in the **`basis`** dictionary. In the following example, the `6-31G` basis set will be assigned to the atom `H1`, but the `STO-3G` basis will be used for the atom `H2`:

> mol.atom = '8 0 0 0; h1 0 1 0; H2 0 0 1' mol.basis = {'O': 'sto-3g', 'H': 'sto3g', 'H1': '6-31G'}

See examples/gto/04-input_basis.py [https://github.com/pyscf/pyscf/tree/master/examples/gto/04-input_basis.py] for more examples.

Basis format

Basis data can be a text file or a python file.

The text file should store the basis data in NWChem format. Most basis in PySCF were downloaded from https://www.basissetexchange.org/ [https://www.basissetexchange.org/] . Some basis (mostly the cc-pV*Z basis) were downloaded with the option “optimize general contractions” checked.

The python basis format stores the basis in the internal format which looks:

```
[[angular, kappa, [[exp, c_1, c_2, ..],
                   [exp, c_1, c_2, ..],
                   ... ]],
 [angular, kappa, [[exp, c_1, c_2, ..],
                   [exp, c_1, c_2, ..]
                   ... ]]]
```

The list *[angular, kappa, [[exp, c, …]]]* defines the angular momentum of the basis, the kappa value, the Gaussian exponents and basis contraction coefficients. *kappa* can have value $-l-1$ (corresponding to spinors with $j=l+1/2$), l (corresponding to spinors with $j=l-1/2$) or 0. When kappa is 0, both types of spinors are assumed in the basis. A few basis for relativistic calculations (e.g. Dyall basis) were saved in this format.

Ordering of basis functions

GTO basis functions are stored in the following order: (1) atoms, (2) angular

momentum, (3) shells, (4) spherical harmonics. This means that basis functions are first

grouped in terms of the atoms they are assigned to. On each atom, basis functions are grouped according to their angular momentum. For each value of the angular momentum, the individual shells are sorted from inner shells to outer shells, that is, from large exponents to small exponents. A shell can be a real atomic shell, formed as a linear combination of many Gaussians, or just a single primitive Gaussian function that may have several angular components. In each shell, the spherical parts of the GTO basis follow the Condon-Shortley convention, with the ordering (and phase) given in *the Wikipedia table of spherical harmonics <https://en.wikipedia.org/wiki/Table_of_spherical_harmonics>*, except for the *p* functions for which the order of *px*, *py*, *pz* is used instead of the order`py`, *pz*, *px* used in the table above.

Short notations are used for basis functions of *s*, *p* and *d* shells. We use the label *z^2* for the *Lz=0* component of *d* function as the short name of *3z^2 - r^2*. For example, after applying all the rules above, we have the following cc-pVTZ basis functions for carbon atom:

Atom Id	Shell	Angular momentum	Spherical part
0 C	1	s	
0 C	2	s	
0 C	3	s	
0 C	4	s	
0 C	2	p	x
0 C	2	p	y
0 C	2	p	z
0 C	3	p	x
0 C	3	p	y
0 C	3	p	z
0 C	4	p	x
0 C	4	p	y
0 C	4	p	z
0 C	3	d	xy
0 C	3	d	yz
0 C	3	d	z^2
0 C	3	d	xz
0 C	3	d	x2-y2

The order of Cartesian GTOs is generated by the code below:

```
for lx in reversed(range(l + 1)):
  for ly in reversed(range(l + 1 - lx)):
    lz = l - lx - ly
    basis = 'x' * lx + 'y' * ly + 'z' * lz
```

For example, the Cartesian *d* functions are ordered as *xx*, *xy*, *xz*, *yy*, *yz*, *zz*.

> The ordering of the basis functions can be verified with the method **`Mole.ao_labels()`**.

ECP

Effective core potentials (ECPs) can be specified with the attribute **`Mole.ecp`**. Scalar type ECPs are available for all molecular and crystal methods. The built-in scalar ECP datasets include

0 C	4	d	xy
0 C	4	d	[illegible]
0 C	4	d	xz
0 C	4	d	x2-y2
0 C	4	f	-3
0 C	4	f	-2
0 C	4	f	-1
0 C	4	f	0
0 C	4	f	1
0 C	4	f	2
0 C	4	f	3

Keyword	Comment
bfd	
cc-pvdz-pp	
cc-pvtz-pp	same to cc-pvdz-pp
cc-pvqz-pp	same to cc-pvdz-pp
cc-pv5z-pp	same to cc-pvdz-pp
crenbl	
crenbs	
def2-svp	
def2-svpd	same to def2-svp
def2-tzvp	same to def2-svp
def2-tzvpd	same to def2-svp
def2-tzvpp	same to def2-svp
def2-tzvppd	same to def2-svp
def2-qzvp	same to def2-svp
def2-qzvpd	same to def2-svp
def2-qzvpp	same to def2-svp
def2-qzvppd	same to def2-svp
lanl2dz	

lanl2tz	
lanl08	
sbkjc	
stuttgart	

ECP parameters can be specified directly in input script using NWChem format. Examples of ECP input can be found in examples/gto/05-input_ecp.py [https://github.com/pyscf/pyscf/tree/master/examples/gto/05-input_ecp.py].

Spin-orbit (SO) ECP integrals can be evaluated using PySCF's integral driver. However, SO-ECPs are not automatically applied to any methods in the current implementation. They need to be added to the core Hamiltonian as shown in the examples examples/gto/20-soc_ecp.py [https://github.com/pyscf/pyscf/tree/master/examples/gto/20-soc_ecp.py] and examples/scf/44-soc_ecp.py [https://github.com/pyscf/pyscf/tree/master/examples/scf/44-soc_ecp.py]. PySCF provides the following SO-ECPs

Keyword	Comment
crenbl	
crenbs	

Note

Be careful with the SO-ECP conventions when inputting them directly in the input script. SO-ECP parameters may take different conventions in different packages. More particularly, the treatment of the factor 2/(2l+1). PySCF assumes that this factor has been multiplied into the SOC parameters. See also relevant discussions in Dirac doc [http://www.diracprogram.org/doc/master/molecule_and_basis/molecule_with_ecp.html] and NWChem doc [https://nwchemgit.github.io/ECP.html].

Point group symmetry

You can also invoke point group symmetry for molecular calculations by setting the attribute **`Mole.symmetry`** to `True`:

```
>>> mol = pyscf.M(
...     atom = 'B 0 0 0; H 0 1 1; H 1 0 1; H 1 1 0',
...     symmetry = True
... )
```

The point group symmetry information is held in the **`Mole`** object. The symmetry module (**`symm`**) of PySCF can detect arbitrary point groups. The detected point group is saved in **`Mole.topgroup`**, and the supported subgroup is saved in **`Mole.groupname`**:

✓
```
>>> print(mol.topgroup)
C3v
>>> print(mol.groupname)
Cs
```

Currently, PySCF supports linear molecular symmetries $D_{\infty h}$ (labelled as `Dooh` in the program) and $C_{\infty v}$ (labelled as `Coov`), the D_{2h} group and its subgroups. Sometimes it is necessary to use a lower symmetry instead of the detected symmetry group. The subgroup symmetry can be specified in by **`Mole.symmetry_subgroup`** and the program will first detect the highest possible symmetry group and then lower the point group symmetry to the specified subgroup:

```
>>> mol = gto.Mole()
>>> mol.atom = 'N 0 0 0; N 0 0 1'
>>> mol.symmetry = True
>>> mol.symmetry_subgroup = C2
>>> mol.build()
>>> print(mol.topgroup)
Dooh
>>> print(mol.groupname)
C2
```

When a particular symmetry is assigned to **`Mole.symmetry`**, the initialization function **`Mole.build()`** will test whether the molecule geometry is subject to the required symmetry. If not, initialization will stop and an error message will be issued:

```
>>> mol = gto.Mole()
>>> mol.atom = 'O 0 0 0; C 0 0 1'
>>> mol.symmetry = 'Dooh'
>>> mol.build()
RuntimeWarning: Unable to identify input symmetry Dooh.
Try symmetry="Coov" with geometry (unit="Bohr")
('O', [0.0, 0.0, -0.809882624813598])
('C', [0.0, 0.0, 1.0798434997514639])
```

Note

`Mole.symmetry_subgroup` has no effects when specific symmetry group is assigned to `Mole.symmetry`.

When symmetry is enabled in the `Mole` object, the point group symmetry information will be used to construct the symmetry adapted orbital basis (see also `symm`). The symmetry adapted orbitals are held in `Mole.symm_orb` as a list of 2D arrays. Each element of the list is an AO (atomic orbital) to SO (symmetry-adapted orbital) transformation matrix of an irreducible representation. The name of the irreducible representations are stored in `Mole.irrep_name` and their internal IDs (see more details in `symm`) are stored in `Mole.irrep_id`:

```
>>> mol = gto.Mole()
>>> mol.atom = 'O 0 0 0; O 0 0 1.2'
>>> mol.spin = 2
>>> mol.symmetry = "D2h"
>>> mol.build()
>>> for s,i,c in zip(mol.irrep_name, mol.irrep_id, mol.symm_orb):
...     print(s, i, c.shape)
Ag 0 (10, 3)
B2g 2 (10, 1)
B3g 3 (10, 1)
B1u 5 (10, 3)
B2u 6 (10, 1)
B3u 7 (10, 1)
```

These symmetry-adapted orbitals are used as basis functions for the following SCF or post-SCF calculations:

```
>>> mf = scf.RHF(mol)
>>> mf.kernel()
converged SCF energy = -147.631655286561
```

and we can check the occupancy of the MOs in each irreducible representation:

```
>>> import numpy
>>> from pyscf import symm
>>> def myocc(mf):
...     mol = mf.mol
...     orbsym = symm.label_orb_symm(mol, mol.irrep_id, mol.symm_orb,
mf.mo_coeff)
...     doccsym = numpy.array(orbsym)[mf.mo_occ==2]
...     soccsym = numpy.array(orbsym)[mf.mo_occ==1]
...     for ir,irname in zip(mol.irrep_id, mol.irrep_name):
...         print('%s, double-occ = %d, single-occ = %d' %
...               (irname, sum(doccsym==ir), sum(soccsym==ir)))
>>> myocc(mf)
Ag, double-occ = 3, single-occ = 0
B2g, double-occ = 0, single-occ = 1
B3g, double-occ = 0, single-occ = 1
B1u, double-occ = 2, single-occ = 0
B2u, double-occ = 1, single-occ = 0
B3u, double-occ = 1, single-occ = 0
```

To label the irreducible representation of given orbitals, `symm.label_orb_symm()` needs the information of the point group symmetry which is initialized in the `Mole` object, including the IDs of irreducible representations (`Mole.irrep_id`) and the symmetry adapted basis `Mole.symm_orb`. For each `irrep_id`, `Mole.irrep_name` gives the associated irrep symbol (A1, B1 …). In the SCF calculation, you can control the symmetry of the wave function by assigning the number of alpha electrons and beta electrons *(alpha,beta)* for the irreps:

```
>>> mf.irrep_nelec = {'B2g': (1,1), 'B3g': (1,1), 'B2u': (1,0), 'B3u':
(1,0)}
>>> mf.kernel()
converged SCF energy = -146.97333043702
```

```
>>> mf.get_irrep_nelec()
{'Ag': (3, 3), 'B2g': (1, 1), 'B3g': (1, 1), 'B1u': (2, 2), 'B2u': (1,
0), 'B3u': (1, 0)}
```

Spin and charge

Charge and the number of unpaired electrons can be assigned to **Mole** object:

```
mol.charge = 1
mol.spin = 1
```

Note

Mole.spin is the number of unpaired electrons *2S*, i.e. the difference between the number of alpha and beta electrons.

These two attributes do not affect any other parameters in the **Mole.build** initialization function. They can be set or modified after the **Mole** object is initialized:

```
>>> mol = gto.Mole()
>>> mol.atom = 'O 0 0 0; h 0 1 0; h 0 0 1'
>>> mol.basis = 'sto-6g'
>>> mol.spin = 2
>>> mol.build()
>>> print(mol.nelec)
(6, 4)
>>> mol.spin = 0
>>> print(mol.nelec)
(5, 5)
```

The attribute **Mole.charge** is the parameter to define the total number of electrons in the system. For custom systems such as the Hubbard lattice model, the total number of electrons needs to be specified directly by setting the attribute **Mole.nelectron**:

```
>>> mol = gto.Mole()
>>> mol.nelectron = 10
```

Other parameters

You can assign more information to the molecular object:

```
mol.nucmod = {'O1': 1} # nuclear charge model: 0-point charge, 1-
Gaussian distribution
mol.mass = {'O1': 18, 'H': 2}  # atomic mass
```

See examples/gto/07-nucmod.py [https://github.com/pyscf/pyscf/tree/master/examples/gto/07-nucmod.py] for more examples of nuclear charge models.

The `Mole` class also defines some global parameters. You can control the print level globally with `verbose`:

✓
```
mol.verbose = 4
```

The print level can be 0 (quiet, no output) to 9 (very noisy). The most useful messages are printed at level 4 (info) and 5 (debug). You can also specify a place where to write the output messages:

✓
```
mol.output = 'path/to/log.txt'
```

If this variable is not assigned, messages will be dumped to `sys.stdout`.

The maximum memory usage can be controlled globally:

✓
```
mol.max_memory = 1000 # MB
```

The default size can also be defined with the shell environment variable *PYSCF_MAX_MEMORY*.

The attributes **output** and **max_memory** can also be assigned from command line:

```
$ python input.py -o /path/to/my_log.txt -m 1000
```

By default, command line has the highest priority, which means the settings in the script will be overwritten by the command line arguments. To make the input parser ignore the command line arguments, you can call the **Mole.build()** with:

```
mol.build(0, 0)
```

The first `0` prevent **build()** dumping the input file. The second `0` prevent **build()** parsing the command line arguments.

Access AO integrals

PySCF uses libcint [https://github.com/sunqm/libcint] library as the AO integral engine. A simple interface function **Mole.intor()** is provided to obtain the one- and two-electron AO integrals:

```
kin = mol.intor('int1e_kin')
vnuc = mol.intor('int1e_nuc')
overlap = mol.intor('int1e_ovlp')
eri = mol.intor('int2e')
```

For a full list of supported AO integrals, see pyscf.gto.moleintor module [../pyscf_api_docs/pyscf.gto.html#module-pyscf.gto.moleintor].