PySCF Documentation

Release 1.4.0

Qiming Sun <osirpt.sun@gmail.com>

Oct 12, 2017

PySCF is a collection of electronic structure programs powered by Python. The package aims to provide a simple, light-weight, and efficient platform for quantum chemistry calculations and code development. The program is developed with the following principles:

- Easy to install, to use, to extend and to be embedded;
- Minimal requirements on libraries (no Boost or MPI) and computing resources (perhaps sacrificing efficiency to reduce I/O);
- 90/10 Python/C (only computational hot spots are written in C);
- 90/10 functional/OOP (unless performance critical, functions are pure).

CHAPTER

ONE

CONTENTS

1.1 An overview of PySCF

Python-based simulations of chemistry framework (PYSCF) is a general-purpose electronic structure platform designed from the ground up to emphasize code simplicity, so as to facilitate new method development and enable flexible computational workflows. The package provides a wide range of tools to support simulations of finite-size systems, extended systems with periodic boundary conditions, low-dimensional periodic systems, and custom Hamiltonians, using mean-field and post-mean-field methods with standard Gaussian basis functions. To ensure ease of extensibility, PYSCF uses the Python language to implement almost all of its features, while computationally critical paths are implemented with heavily optimized C routines. Using this combined Python/C implementation, the package is as efficient as the best existing C or Fortran- based quantum chemistry programs.

1.1.1 Features

- Hartree-Fock (up to ~5000 basis)
  - Non-relativistic restricted open-shell, unrestricted HF
  - Scalar relativistic HF
  - 2-component relativistic HF
  - 4-component relativistic Dirac-Hartree-Fock
  - Density fitting HF
  - Second order SCF
  - General JK contraction function
  - DIIS, EDIIS, ADIIS and second order solver
  - SCF wavefunction stability analysis
  - Generalized Hartree-Fock (GHF)
- DFT (up to ~5000 basis)
  - Non-relativistic restricted, restricted open-shell, unrestricted Kohn-Sham
  - Scalar relativistic DFT
  - Density fitting DFT
  - General XC functional evaluator (for Libxc or XcFun)
  - General AO evaluator
- TDDFT

  - TDHF (and density-fitting TDHF)
  - TDDFT (and density-fitting TDDFT)
  - TDHF gradients
  - TDDFT gradients
- General CASCI/CASSCF solver (up to ~3000 basis)
  - State-average CASCI/CASSCF
  - State-specific CASCI/CASSCF for excited states
  - Multiple roots CASCI
  - Support DMRG as plugin CI solver to do DMRG-CASSCF
  - Support FCIQMC as plugin CI solver to do FCIQMC-CASSCF
  - UHF-based UCASSCF
  - Density-fitting CASSCF
  - DMET-CAS and AVAS active space constructor
- MP2 (up to ~200 occupied, ~2000 virtual orbitals)
  - Canonical MP2
  - Density-fitting MP2
  - MP2-F12
- CCSD (up to ~100 occupied, ~1500 virtual orbitals)
  - canonical RCCSD, UCCSD
  - canonical RCCSD, UCCSD lambda solver
  - RCCSD and UCCSD 1-particle and 2-particle density matrices
  - CCSD gradients
  - EOM-IP/EA/EE-CCSD
  - RCC2
  - Density-fitting CCSD
- CCSD(T)
  - Canonical RCCSD(T) and UCCSD(T)
  - Canonical RCCSD(T) 1- and 2-particle density matrices
  - Canonical RCCSD(T) gradients
- CI
  - RCISD and UCISD
  - RCISD and UCISD 1, 2-particle density matrices
  - Selected-CI
  - Selected-CI 1, 2-particle density matrices
- Full CI
  - Direct-CI solver for spin degenerated Hamiltonian

  - Direct-CI solver for spin non-degenerated Hamiltonian
  - 1, and 2-particle transition density matrices
  - 1, 2, 3, and 4-particle density matrices
  - CI wavefunction overlap
- Gradients
  - Non-relativistic RHF gradients
  - 4-component DHF gradients
  - Non-relativistic DFT gradients
  - Non-relativistic CCSD gradients
  - Non-relativistic TDHF and TDDFT gradients
- Hessian
  - Non-relativistic RHF hessian
  - Non-relativistic RKS hessian
- Properties
  - Non-relativistic RHF, UHF, RKS, UKS NMR shielding
  - 4-component DHF NMR shielding
  - Non-relativistic RHF, UHF spin-spin coupling
  - 4-component DHF spin-spin coupling
  - Non-relativistic UHF, UKS hyperfine coupling
  - 4-component DHF hyperfine coupling
  - Non-relativistic UHF, UKS g-tensor
  - 4-component DHF g-tensor
  - Non-relativistic UHF zero-field splitting
  - Molecular electrostatic potential (MEP)
- MRPT
  - Strongly contracted NEVPT2
  - DMRG-NEVPT2
  - IC-MPS-PT2
- Extended systems with periodic boundary condition
  - gamma point RHF, UHF, RKS, UKS
  - gamma point TDDFT, MP2, CCSD
  - PBC RHF, UHF, RKS, UKS with k-point sampling
  - PBC RCCSD and UCCSD with k-point sampling
  - PBC k-point EOM-IP/EA-CCSD
  - PBC AO integrals
  - PBC MO integral transformation

  - PBC density fitting
  - Smearing for mean-field calculation
  - Low-dimensional (0D, 1D, 2D) PBC systems
  - TDHF and TDDFT with k-point sampling
- AO integrals
  - Interface to call Libcint library
  - 1-electron real-GTO and spinor-GTO integrals
  - 2-electron real-GTO and spinor-GTO integrals
  - 3-center 1-electron real-GTO and spinor-GTO integrals
  - 3-center 2-electron real-GTO and spinor-GTO integrals
  - General basis value evaluator (for numeric integration)
  - PBC 1-electron integrals
  - PBC 2-electron integrals
  - F12 integrals
- MO integrals
  - 2-electron integral transformation for any integrals provided by Libcint library
  - Support for 4-index integral transformation with 4 different orbitals
  - PBC 2-electron MO integrals
- Localizer
  - Boys
  - Edmiston
  - Meta-Lowdin
  - Natural atomic orbital (NAO)
  - Intrinsic atomic orbital (IAO)
  - Pipek-Mezey
- Geometry optimization
  - RHF, RKS, RCCSD with pyberny geometry optimizer
- D2h symmetry and linear molecule symmetry
  - Molecule symmetry detection
  - Symmetry adapted basis
  - Label orbital symmetry on the fly
  - Hot update symmetry information
  - Function to symmetrize given orbital space
- Tools
  - fcidump
  - molden

  - cubegen
  - Molpro XML reader
- Interface to integral package Libcint
- Interface to DMRG CheMPS2
- Interface to DMRG Block
- Interface to FCIQMC NECI
- Interface to XC functional library XCFun
- Interface to XC functional library Libxc

PySCF Python-based simulations of chemistry framework

How to use

There are two ways to access the documentation: the docstrings come with the code, and an online program reference, available from http://www.sunqm.net/pyscf/index.html

We recommend the enhanced Python interpreter IPython and the web-based Python IDE Ipython notebook to try out the package:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.2', basis='cc-pvdz')
>>> mol.apply(scf.RHF).run()
converged SCF energy = -1.06111199785749
-1.06111199786
```

Pure function and Class

Class are designed to hold only the final results and the control parameters such as maximum number of iterations, convergence threshold, etc. The intermediate status are not saved in the class. If the .kernel() function is finished without any errors, the solution will be saved in the class (see documentation).

Most useful functions are implemented at module level, and can be accessed in both class and module, e.g. `scf.hf.get_jk(mol,dm)` and `SCF(mol).get_jk(mol,dm)` have the same functionality. As a result, most functions and class are **pure**, i.e. no status are saved, and the argument are not changed inplace. Exceptions (destructive functions and methods) are suffixed with underscore in the function name, eg `mcscf.state_average_(mc)` changes the attribute of its argument `mc`

Stream functions

For most methods, there are three stream functions to pipe computing stream:

1 `.set` function to update object attributes, eg `mf = scf.RHF(mol).set(conv_tol=1e-5)` is identical to proceed in two steps `mf = scf.RHF(mol); mf.conv_tol=1e-5`

2 `.run` function to execute the kernel function (the function arguments are passed to kernel function). If keyword arguments is given, it will first call `.set` function to update object attributes then execute the kernel function. Eg `mf = scf.RHF(mol).run(dm_init,conv_tol=1e-5)` is identical to three steps `mf = scf.RHF(mol); mf.conv_tol=1e-5; mf.kernel(dm_init)`

3 `.apply` function to apply the given function/class to the current object (function arguments and keyword arguments are passed to the given function). Eg `mol.apply(scf.RHF).run().apply(mcscf.CASSCF,6,4,frozen=4)` is identical to `mf = scf.RHF(mol); mf.kernel(); mcscf.CASSCF(mf,6,4,frozen=4)`

1.2 Tutorial

1.2.1 Quick setup

The prerequisites of PySCF include cmake, numpy, scipy, and h5py. On the Ubuntu host, you can quickly install them:

```
$ sudo apt-get install python-h5py python-scipy cmake
```

Then download the latest version of pyscf and build C extensions in `pyscf/lib`:

```
$ git clone https://github.com/sunqm/pyscf
$ cd pyscf/lib
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Finally, update the Python runtime path `PYTHONPATH` (assuming pyscf is put in /home/abc, replace it with your own path):

```
$ echo 'export PYTHONPATH=/home/abc:$PYTHONPATH' >> ~/.bashrc
$ source ~/.bashrc
```

To ensure the installation is successed, start a Python shell, and type:

```
>>> import pyscf
```

If you got errors like:

```
ImportError: No module named pyscf
```

It's very possible that you put `/home/abc/pyscf` in `PYTHONPATH`. You need to remove the `/pyscf` in that string and try `import pyscf` in the python shell again.

Note: The quick setup does not provide the best performance. Please see *Installation* for the installation with optimized libraries.

1.2.2 A simple example

Here is an example to run HF calculation for hydrogen molecule:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.2', basis='ccpvdz')
>>> mf = scf.RHF(mol)
>>> mf.kernel()
```

```
converged SCF energy = -1.06111199785749
-1.06111199786
```

1.2.3 Initializing a molecule

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

```
>>> 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 function `Mole.M()`. This function pass all arguments to `Mole.build()`:

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

Either way, you may have noticed two keywords `atom` and `basis`. They are used to hold the molecular geometry and basis sets.

Geometry

Molecular geometry can be input in Cartesian format:

```
>>> mol = gto.Mole()
>>> mol.atom = '''O 0, 0, 0
... H   0  1  0; H 0, 0, 1'''
```

The atoms in the molecule are represented by an element symbol plus three numbers for coordinates. Different atoms should be separated by `;` or line break. In the same atom, `,` can be used to separate different items. Z-matrix input format is also supported by the input parser:

```
>>> 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 line break. If you need to label an atom to distinguish it from the rest, you can prefix or suffix number or special characters `1234567890~!@#$%^&*()_+.?:<>[]{}|` (except `,` and `;`) to an atomic symbol. With this decoration, you can specify different basis sets, or masses, or nuclear models for different atoms:

```
>>> mol = gto.Mole()
>>> mol.atom = '''8 0 0 0; h:1 0 1 0; H@2 0 0'''
>>> 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]]]
```

Basis set

The simplest way is to assign a string of basis name to `mol.basis`:

```
mol.basis = 'sto3g'
```

This input will apply the specified basis set to all atoms. The basis name in the string is case insensitive. White space, dash and underscore in the basis name are all ignored. If different basis sets are required for different elements, a python `dict` can be assigned to the basis attribute:

```
mol.basis = {'O': 'sto3g', 'H': '6-31g'}
```

You can find more examples in section input_basis and in the file `examples/gto/04-input_basis.py`.

Other parameters

You can assign more informations to the molecular object:

```
mol.symmetry = 1
mol.charge = 1
mol.spin = 1
mol.nucmod = {'O1': 1}
mol.mass = {'O1': 18, 'H': 2}
```

Note: `Mole.spin` is *2S*, the alpha and beta electron number difference.

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

```
mol.verbose = 4
```

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

```
mol.output = 'path/to/my_log.txt'
```

Without assigning this variable, messages will be dumped to `sys.stdout`. You can control the maximum memory usage globally:

```
mol.max_memory = 1000 # MB
```

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

`output` and `max_memory` can be assigned from command line:

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

1.2.4 Initializing a crystal

Initialization a crystal unit cell is very similar to the initialization molecular object. Here, *pyscf.pbc.gto.Cell* class should be used instead of the `pyscf.gto.Mole` class:

```
>>> 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 crystal initialization requires an extra parameter `cell.a` which represents the lattice vectors. In the above example, we specified `cell.pseudo` for the pseudo-potential of the system which is an optional parameter. The input format of basis set is the same to that of `Mole` object. The other attributes of `Mole` object such as `verbose`, `max_memory`, `spin` can also be used in the crystal systems. More details of the crystal `Cell` object and the relevant input parameters are documented in *pbc.gto — Crystal cell structure*.

1D and 2D systems

PySCF PBC module supports the low-dimensional PBC systems. You can initialize the attribute `cell.dimension` to specify the dimension of the system:

```
>>> from pyscf.pbc import gto
>>> cell = gto.Cell()
>>> cell.atom = '''H  0 0 0; H 1 1 0'''
>>> cell.basis = 'sto3g'
>>> cell.dimension = 2
>>> cell.a = numpy.eye(3) * 2
>>> cell.build()
```

When `cell.dimension` is specified, a vacuum of infinite size will be applied on certain dimension(s). More specifically, when `cell.dimension` is 2, the z-direction will be treated as infinite large and the xy-plane constitutes the periodic surface. When `cell.dimension` is 1, y and z axes are treated as vacuum thus wire is placed on the x axis. When `cell.dimension` is 0, all three directions are vacuum. The PBC system is actually the same to the molecular system.

1.2.5 HF, MP2, MCSCF

Hartree-Fock

Now we are ready to study electronic structure theory with pyscf. Let's take oxygen molecule as the first example:

```
>>> from pyscf import gto
>>> mol = gto.Mole()
>>> mol.verbose = 5
>>> mol.output = 'o2.log'
>>> mol.atom = 'O 0 0 0; O 0 0 1.2'
>>> mol.basis = 'ccpvdz'
>>> mol.build()
```

Apply non-relativistic Hartree-Fock:

```
>>> from pyscf import scf
>>> m = scf.RHF(mol)
>>> print('E(HF) = %g' % m.kernel())
E(HF) = -149.544214749
```

The ground state of oxygen molecule should be triplet. So we change the spin to 2 (2 more alpha electrons than beta electrons):

```
>>> o2_tri = mol.copy()
>>> o2_tri.spin = 2
>>> o2_tri.build(0, 0)  # two "0"s to prevent dumping input and parsing command line
>>> rhf3 = scf.RHF(o2_tri)
>>> print(rhf3.kernel())
-149.609461122
```

Run UHF:

```
>>> uhf3 = scf.UHF(o2_tri)
>>> print(uhf3.scf())
-149.628992314
>>> print('S^2 = %f, 2S+1 = %f' % uhf3.spin_square())
S^2 = 2.032647, 2S+1 = 3.021686
```

where we called `mf.scf()`, which is an alias name of `mf.kernel`. You can impose symmetry:

```
>>> o2_sym = mol.copy()
>>> o2_sym.spin = 2
>>> o2_sym.symmetry = 1
>>> o2_sym.build(0, 0)
>>> rhf3_sym = scf.RHF(o2_sym)
>>> print(rhf3_sym.kernel())
-149.609461122
```

Here we rebuild the molecule because we need to initialize the point group symmetry information, symmetry adapted orbitals. We can check the occupancy for each irreducible representations:

```
>>> import numpy
>>> from pyscf import symm
>>> def myocc(mf):
...     mol = mf.mol
...     irrep_id = mol.irrep_id
...     so = mol.symm_orb
...     orbsym = symm.label_orb_symm(mol, irrep_id, so, mf.mo_coeff)
...     doccsym = numpy.array(orbsym)[mf.mo_occ==2]
...     soccsym = numpy.array(orbsym)[mf.mo_occ==1]
...     for ir,irname in enumerate(mol.irrep_name):
...         print('%s, double-occ = %d, single-occ = %d' %
...               (irname, sum(doccsym==ir), sum(soccsym==ir)))
>>> myocc(rhf3_sym)
Ag, double-occ = 3, single-occ = 0
B1g, double-occ = 0, single-occ = 0
B2g, double-occ = 0, single-occ = 1
B3g, double-occ = 0, single-occ = 1
Au, double-occ = 0, single-occ = 0
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 are initialized in `mol` object, including the *id* 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 some irreps:

```
>>> rhf3_sym.irrep_nelec = {'B2g': (1,1), 'B3g': (1,1), 'B2u': (1,0), 'B3u': (1,0)}
>>> rhf3_sym.kernel()
>>> print(rhf3_sym.kernel())
-148.983117701
>>> rhf3_sym.get_irrep_nelec()
{'Ag' : (3, 3), 'B1g': (0, 0), 'B2g': (1, 1), 'B3g': (1, 1), 'Au' : (0, 0), 'B1u':␣
↪(1, 0), 'B2u': (0, 1), 'B3u': (1, 0)}
```

More informations of the calculation can be found in the output file `o2.log`.

MP2 and MO integral transformation

Next, we compute the correlation energy with `mp.mp2`:

```
>>> from pyscf import mp
>>> mp2 = mp.MP2(m)
>>> print('E(MP2) = %.9g' % mp2.kernel()[0])
E(MP2) = -0.379359288
```

This is the correlation energy of singlet ground state. For the triplet state, we can write a function to compute the correlation energy

$$E_{corr} = \frac{1}{4}\sum_{ijab}\frac{\langle ij||ab\rangle\langle ab||ij\rangle}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b}$$

```
def myump2(mf):
    import numpy
    from pyscf import ao2mo
    # As UHF objects, mo_energy, mo_occ, mo_coeff are two-item lists
    # (the first item for alpha spin, the second for beta spin).
    mo_energy = mf.mo_energy
    mo_occ = mf.mo_occ
    mo_coeff = mf.mo_coeff
    o = numpy.hstack((mo_coeff[0][:,mo_occ[0]>0] ,mo_coeff[1][:,mo_occ[1]>0]))
    v = numpy.hstack((mo_coeff[0][:,mo_occ[0]==0],mo_coeff[1][:,mo_occ[1]==0]))
    eo = numpy.hstack((mo_energy[0][mo_occ[0]>0] ,mo_energy[1][mo_occ[1]>0]))
    ev = numpy.hstack((mo_energy[0][mo_occ[0]==0],mo_energy[1][mo_occ[1]==0]))
    no = o.shape[1]
    nv = v.shape[1]
    noa = sum(mo_occ[0]>0)
    nva = sum(mo_occ[0]==0)
    eri = ao2mo.general(mf.mol, (o,v,o,v)).reshape(no,nv,no,nv)
    eri[:noa,nva:] = eri[noa:,:nva] = eri[:,:,:noa,nva:] = eri[:,:,noa:,:nva] = 0
    g = eri - eri.transpose(0,3,2,1)
    eov = eo.reshape(-1,1) - ev.reshape(-1)
    de = 1/(eov.reshape(-1,1) + eov.reshape(-1)).reshape(g.shape)
    emp2 = .25 * numpy.einsum('iajb,iajb,iajb->', g, g, de)
    return emp2
```

```
>>> print('E(UMP2) = %.9g' % myump2(uhf3))
-0.346926068
```

In this example, we concatenate α and β orbitals to mimic the spin-orbitals. After integral transformation, we zeroed out the integrals of different spin. Here, the `ao2mo` module provides the general 2-electron MO integral transformation. Using this module, you are able to do *arbitrary* integral transformation for *arbitrary* integrals. For example, the following code gives the `(ov|vv)` type integrals:

```
>>> from pyscf import ao2mo
>>> import h5py
>>> mocc = m.mo_coeff[:,m.mo_occ>0]
>>> mvir = m.mo_coeff[:,m.mo_occ==0]
>>> ao2mo.general(mol, (mocc,mvir,mvir,mvir), 'tmp.h5', compact=False)
>>> feri = h5py.File('tmp.h5')
>>> ovvv = numpy.array(feri['eri_mo'])
>>> print(ovvv.shape)
(160, 400)
```

We pass `compact=False` to `ao2mo.general()` to prevent the function using the permutation symmetry between the virtual-virtual pair of `|vv)`. So the shape of `ovvv` corresponds to 8 occupied orbitals by 20 virtual orbitals for electron 1 `(ov|` and 20 by 20 for electron 2 `|vv)`. In the following example, we transformed the analytical gradients of 2-electron integrals

$$\langle(\frac{\partial}{\partial R}\varphi_i)\varphi_k|\varphi_j\varphi_l\rangle = \int \frac{\frac{\partial\varphi_i(r_1)}{\partial R}\varphi_j(r_1)\varphi_k(r_2)\varphi_l(r_2)}{|r_1-r_2|}dr_1dr_2$$

```
>>> nocc = mol.nelectron // 2
>>> co = mf.mo_coeff[:,:nocc]
>>> cv = mf.mo_coeff[:,nocc:]
>>> nvir = cv.shape[1]
>>> eri = ao2mo.general(mol, (co,cv,co,cv), intor='int2e_ip1_sph', comp=3)
>>> eri = eri.reshape(3, nocc, nvir, nocc, nvir)
>>> print(eri.shape)
(3, 8, 20, 8, 20)
```

CASCI and CASSCF

The two classes `mcscf.CASCI` and `mcscf.CASSCF` provided by `mcscf` have the same initialization interface:

```
>>> from pyscf import mcscf
>>> mc = mcscf.CASCI(m, 4, 6)
>>> print('E(CASCI) = %.9g' % mc.casci()[0])
E(CASCI) = -149.601051
>>> mc = mcscf.CASSCF(m, 4, 6)
>>> print('E(CASSCF) = %.9g' % mc.kernel()[0])
E(CASSCF) = -149.613191
```

In this example, the CAS space is (6e, 4o): the third argument for CASCI/CASSCF is the size of CAS space; the fourth argument is the number of electrons. By default, the CAS solver determines the alpha-electron number and beta-electron number based on the attribute `Mole.spin`. In the above example, the number of alpha electrons is equal to the number of beta electrons, since the `mol` object is initialized with `spin=0`. The spin multiplicity of the CASSCF/CASCI solver can be changed by the fourth argument:

```
>>> mc = mcscf.CASSCF(m, 4, (4,2))
>>> print('E(CASSCF) = %.9g' % mc.kernel()[0])
E(CASSCF) = -149.609461
>>> print('S^2 = %.7f, 2S+1 = %.7f' % mcscf.spin_square(mc))
S^2 = 2.0000000, 2S+1 = 3.0000000
```

The two integers in the tuple represent the number of alpha and beta electrons. Although it is a triplet state, the solution might not be correct since the CASSCF is based on the incorrect singlet HF ground state. Starting from the ROHF ground state, we have:

```
>>> mc = mcscf.CASSCF(rhf3, 4, 6)
>>> print('E(CASSCF) = %.9g' % mc.kernel()[0])
E(CASSCF) = -149.646746
```

The energy is lower than the RHF initial guess. .. We can also use the UHF ground .. state to start a CASSCF calculation:: .. .. >>> mc = mcscf.CASSCF(uhf3, 4, 6) .. >>> print('E(CASSCF) = %.9g' % mc.kernel()[0]) .. E(CASSCF) = -149.661324 .. >>> print('S^2 = %.7f, 2S+1 = %.7f' % mcscf.spin_square(mc)) .. S^2 = 3.9713105, 2S+1 = 4.1091656 .. .. Woo, the total energy is even lower. But the spin is contaminated.

1.2.6 Restore an old calculation

There is no *restart* mechanism available in PySCF package. Calculations can be "restarted" by the proper initial guess. For SCF, the initial guess can be prepared in many ways. One is to read the `chkpoint` file which is generated in the previous or other calculations:

```
>>> from pyscf import scf
>>> mf = scf.RHF(mol)
>>> mf.chkfile = '/path/to/chkfile'
>>> mf.init_guess = 'chkfile'
>>> mf.kernel()
```

`/path/to/chkfile` can be found in the output in the calculation (if mol.verbose >= 4, the filename of the chkfile will be dumped in the output). By setting `chkfile` and `init_guess`, the SCF module can read the molecular orbitals from the given `chkfile` and rotate them to representation of the required basis. The example `examples/scf/15-initial_guess.py` records other methods to generate SCF initial guess.

Initial guess can be fed to the calculation directly. For example, we can read the initial guess form a chkfile and achieve the same effects as the on in the previous example:

```
>>> from pyscf import scf
>>> mf = scf.RHF(mol)
>>> dm = scf.hf.from_chk(mol, '/path/to/chkfile')
>>> mf.kernel(dm)
```

`scf.hf.from_chk()` reads the chkpoint file and generates the corresponding density matrix represented in the required basis.

Initial guess `chkfile` is not limited to the calculation based on the same molecular and same basis set. One can first do a cheap SCF (with small basis sets) or a model SCF (dropping a few atoms, or charged system), then use `scf.hf.from_chk()` to project the results to the target basis sets.

To restart a CASSCF calculation, you need prepare either CASSCF orbitals or CI coefficients (not that useful unless doing a DMRG-CASSCF calculation) or both. For example:

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

```
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

import tempfile
from pyscf import gto, scf, mcscf
from pyscf import lib

'''
Restart CASSCF from previous calculation.

There is no "restart" keyword for CASSCF solver.  The CASSCF solver is
completely controlled by initial guess.  So we can mimic the restart feature
by providing proper initial guess from previous calculation.

We need assign the .chkfile a string to indicate the file where to save the
CASSCF intermediate results.  Then we can "restart" the calculation from the
intermediate results.
'''

tmpchk = tempfile.NamedTemporaryFile()

mol = gto.Mole()
mol.atom = 'C 0 0 0; C 0 0 1.2'
mol.basis = 'ccpvdz'
mol.build()

mf = scf.RHF(mol)
mf.kernel()

mc = mcscf.CASSCF(mf, 6, 6)
mc.chkfile = tmpchk.name
mc.max_cycle_macro = 1
mc.kernel()

######################################################################
#
# Assuming the CASSCF was interrupted.  Intermediate data were saved in
# tmpchk file.  Here we read the chkfile to restart the previous calculation.
#
######################################################################
mol = gto.Mole()
mol.atom = 'C 0 0 0; C 0 0 1.2'
mol.basis = 'ccpvdz'
mol.build()

mc = mcscf.CASSCF(scf.RHF(mol), 6, 6)
mo = lib.chkfile.load(tmpchk.name, 'mcscf/mo_coeff')
mc.kernel(mo)
```

1.2.7 Access AO integrals

molecular integrals

PySCF uses Libcint library as the AO integral engine. It provides simple interface function `getints_by_shell()` to evaluate integrals. The following example evaluates 3-center 2-electron integrals with this function:

```
import numpy
from pyscf import gto, scf, df
mol = gto.M(atom='O 0 0 0; h 0 -0.757 0.587; h 0 0.757 0.587', basis='cc-pvdz')
auxmol = gto.M(atom='O 0 0 0; h 0 -0.757 0.587; h 0 0.757 0.587', basis='weigend')
pmol = mol + auxmol
nao = mol.nao_nr()
naux = auxmol.nao_nr()
eri3c = numpy.empty((nao,nao,naux))
pi = 0
for i in range(mol.nbas):
    pj = 0
    for j in range(mol.nbas):
        pk = 0
        for k in range(mol.nbas, mol.nbas+auxmol.nbas):
            shls = (i, j, k)
            buf = pmol.intor_by_shell('int3c2e_sph', shls)
            di, dj, dk = buf.shape
            eri3c[pi:pi+di,pj:pj+dj,pk:pk+dk] = buf
            pk += dk
        pj += dj
    pi += di
```

Here we load the Weigend density fitting basis to `auxmol` and append the basis to normal orbital basis which was initialized in `mol`. In the result `pmol` object, the first `mol.nbas` shells are the orbital basis and the next `auxmol.nbas` are auxiliary basis. The three nested loops run over all integrals for the three index integral *(ij|K)*. Similarly, we can compute the two center Coulomb integrals:

```
eri2c = numpy.empty((naux,naux))
pk = 0
for k in range(mol.nbas, mol.nbas+auxmol.nbas):
    pl = 0
    for l in range(mol.nbas, mol.nbas+auxmol.nbas):
        shls = (k, l)
        buf = pmol.intor_by_shell('int2c2e_sph', shls)
        dk, dl = buf.shape
        eri2c[pk:pk+dk,pl:pl+dl] = buf
        pl += dl
    pk += dk
```

Now we can use the two-center integrals and three-center integrals to implement the density fitting Hartree-Fock code.

```
def get_vhf(mol, dm, *args, **kwargs):
    naux = eri2c.shape[0]
    nao = mol.nao_nr()
    rho = numpy.einsum('ijp,ij->p', eri3c, dm)
    rho = numpy.linalg.solve(eri2c, rho)
    jmat = numpy.einsum('p,ijp->ij', rho, eri3c)
    kpj = numpy.einsum('ijp,jk->ikp', eri3c, dm)
    pik = numpy.linalg.solve(eri2c, kpj.reshape(-1,naux).T)
    kmat = numpy.einsum('pik,kjp->ij', pik.reshape(naux,nao,nao), eri3c)
    return jmat - kmat * .5

mf = scf.RHF(mol)
mf.verbose = 0
mf.get_veff = get_vhf
print('E(DF-HF) = %.12f, ref = %.12f' % (mf.kernel(), scf.density_fit(mf).kernel()))
```

Your screen should output

E(DF-HF) = -76.025936299702, ref = -76.025936299702

Evaluating the integrals with nested loops and `mol.intor_by_shell()` method is inefficient. It is preferred to load integrals in bulk and this can be done with `mol.intor()` method:

```
eri2c = auxmol.intor('int2c2e_sph')
eri3c = pmol.intor('int3c2e_sph', shls_slice=(0,mol.nbas,0,mol.nbas,mol.nbas,mol.
↪nbas+auxmol.nbas))
eri3c = eri3c.reshape(mol.nao_nr(), mol.nao_nr(), -1)
```

`mol.intor()` method can be used to evaluate one-electron integrals, two-electron integrals:

```
hcore = mol.intor('int1e_nuc_sph') + mol.intor('int1e_kin_sph')
overlap = mol.intor('int1e_ovlp_sph')
eri = mol.intor('int2e_sph')
```

There is a long list of supported AO integrals. See *moleintor*.

PBC AO integrals

`mol.intor()` can only be used to evaluate the integrals with open boundary conditions. When the periodic boundary conditions of crystal systems are studied, you need to use `pbc.Cell.pbc_intor()` function to evaluate the integrals of short-range operators, such as the overlap, kinetic matrix:

```
from pyscf.pbc import gto
cell = gto.Cell()
cell.atom = 'H 0 0 0; H 1 1 1'
cell.a = numpy.eye(3) * 2.
cell.build()
overlap = cell.pbc_intor('int1e_ovlp_sph')
```

By default, `pbc.Cell.pbc_intor()` function returns the Γ-point integrals. If k-points are specified, function `pbc.Cell.pbc_intor()` can also evaluate the k-point integrals:

```
kpts = cell.make_kpts([2,2,2])  # 8 k-points
overlap = cell.pbc_intor('int1e_ovlp_sph', kpts=kpts)
```

Note: `pbc.Cell.pbc_intor()` can only be used to evaluate the short-range integrals. PBC density fitting method has to be used to compute the long-range operator such as nuclear attraction integrals, Coulomb integrals.

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

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

See also *pbc.df — PBC denisty fitting* for more details of the PBC density fitting module.

1.2.8 Other features

Density fitting

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

from pyscf import gto
from pyscf import scf

'''
Density fitting method by decorating the scf object with scf.density_fit function.

There is no flag to control the program to do density fitting for 2-electron
integration.  The way to call density fitting is to decorate the existed scf
object with scf.density_fit function.

NOTE scf.density_fit function generates a new object, which works exactly the
same way as the regular scf method.  The density fitting scf object is an
independent object to the regular scf object which is to be decorated.  By
doing so, density fitting can be applied anytime, anywhere in your script
without affecting the exsited scf object.

See also:
examples/df/00-with_df.py
examples/df/01-auxbasis.py
'''

mol = gto.Mole()
mol.build(
    verbose = 0,
    atom = '''8  0  0.     0
              1  0  -0.757 0.587
              1  0  0.757  0.587''',
    basis = 'ccpvdz',
)

mf = scf.density_fit(scf.RHF(mol))
energy = mf.kernel()
print('E = %.12f, ref = -76.026744737355' % energy)

#
# Stream style: calling .density_fit method to return a DF-SCF object.
#
mf = scf.RHF(mol).density_fit()
energy = mf.kernel()
print('E = %.12f, ref = -76.026744737355' % energy)

#
# By default optimal auxiliary basis (if possible) or even-tempered gaussian
# functions are used fitting basis.  You can assign with_df.auxbasis to change
# the change the fitting basis.
#
mol.spin = 1
mol.charge = 1
mol.build(0, 0)
mf = scf.UKS(mol).density_fit()
mf.with_df.auxbasis = 'cc-pvdz-jkfit'
energy = mf.kernel()
print('E = %.12f, ref = -75.390366559552' % energy)
```

Customizing Hamiltonian

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

import numpy
from pyscf import gto, scf, ao2mo

'''
Customizing Hamiltonian for SCF module.

Three steps to define Hamiltonian for SCF:
1. Specify the number of electrons. (Note mole object must be "built" before doing
 ↪this step)
2. Overwrite three attributes of scf object
    .get_hcore
    .get_ovlp
    ._eri
3. Specify initial guess (to overwrite the default atomic density initial guess)

Note you will see warning message on the screen:

        overwrite keys get_ovlp get_hcore of <class 'pyscf.scf.hf.RHF'>
'''

mol = gto.M()
n = 10
mol.nelectron = n

mf = scf.RHF(mol)
h1 = numpy.zeros((n,n))
for i in range(n-1):
    h1[i,i+1] = h1[i+1,i] = -1.0
h1[n-1,0] = h1[0,n-1] = -1.0  # PBC
eri = numpy.zeros((n,n,n,n))
for i in range(n):
    eri[i,i,i,i] = 4.0

mf.get_hcore = lambda *args: h1
mf.get_ovlp = lambda *args: numpy.eye(n)
# ao2mo.restore(8, eri, n) to get 8-fold permutation symmetry of the integrals
# ._eri only supports the two-electron integrals in 4-fold or 8-fold symmetry.
mf._eri = ao2mo.restore(8, eri, n)

mf.kernel()
```

Symmetry in CASSCF

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

```
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

from pyscf import gto, scf, mcscf

'''
Symmetry is not immutable

In PySCF, symmetry is not built-in data structure.  Orbitals are stored in C1
symmetry.  The irreps and symmetry information are generated on the fly.
We can switch on symmetry for CASSCF solver even the Hartree-Fock is not
optimized with symmetry.
'''

mol = gto.Mole()
mol.build(
    atom = [['O' , (0. , 0.     , 0.)],
            [1   , (0. , -0.757 , 0.587)],
            [1   , (0. , 0.757  , 0.587)]],
    basis = 'cc-pvdz',
)
mf = scf.RHF(mol)
mf.kernel()

mol.build(0, 0, symmetry = 'C2v')
mc = mcscf.CASSCF(mf, 6, 8)
mc.kernel()
```

1.3 Installation

We provide three ways to install PySCF package.

1.3.1 Installation with conda

If you have Anaconda environment, PySCF package can be installed with:

```
$ conda install -c pyscf pyscf
```

1.3.2 Installation with pip

You have to first install the dependent libraries (due to the missing of build-time dependency in pip PEP 518):

```
$ pip install numpy scipy h5py
```

Then install PySCF:

```
$ pip install pyscf
```

Note: libxc library is not available in the PyPI repository. pyscf.dft module is not working unless the libxc library was installed in the system. You can download libxc library from http://octopus-code.org/wiki/Libxc:download. You

need to add –enable-shared when compiling the libxc library. Before calling pip, the path where the libxc library is installed needs to be added to the environment variable PYSCF_INC_DIR

1.3.3 Manual installation from github repo

You can manually install PySCF from the PySCF github repo. Manual installation requires cmake, numpy, scipy and h5py libraries. You can download the latest PySCF version (or the development branch) from github:

```
$ git clone https://github.com/sunqm/pyscf
$ cd pyscf
$ git checkout dev  # optional if you'd like to try out the development branch
```

Build the C extensions in pyscf/lib:

```
$ cd pyscf/lib
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This will automatically download the analytical GTO integral library libcint and the DFT exchange correlation functional libraries libxc and xcfun. Finally, to make Python able to find the pyscf package, add the top-level pyscf directory (not the pyscf/pyscf subdirectory) to PYTHONPATH. For example, if pyscf is installed in /opt, PYTHONPATH should be like:

```
export PYTHONPATH=/opt/pyscf:$PYTHONPATH
```

To ensure the installation is successful, start a Python shell, and type:

```
>>> import pyscf
```

For Mac OS X/macOS, you may get an import error if your OS X/macOS version is 10.11 or later:

```
OSError: dlopen(xxx/pyscf/pyscf/lib/libcgto.dylib, 6): Library not loaded: libcint.3.
↪0.dylib
Referenced from: xxx/pyscf/pyscf/lib/libcgto.dylib
Reason: unsafe use of relative rpath libcint.3.0.dylib in xxx/pyscf/pyscf/lib/libcgto.
↪dylib with restricted binary
```

This is caused by the RPATH. It can be fixed by running the script pyscf/lib/_runme_to_fix_dylib_osx10.11.sh in pyscf/lib directory:

```
cd pyscf/lib
sh _runme_to_fix_dylib_osx10.11.sh
```

Note: RPATH has been built in the dynamic library. This may cause library loading error on some systems. You can run pyscf/lib/_runme_to_remove_rpath.sh to remove the rpath code from the library head. Another workaround is to set -DCMAKE_SKIP_RPATH=1 and -DCMAKE_MACOSX_RPATH=0 in cmake command line. When the RPATH was removed, you need to add pyscf/lib and pyscf/lib/deps/lib in LD_LIBRARY_PATH.

A useful last step is to set the scratch directory. The default scratch directory of PySCF is controlled by environment variable PYSCF_TMPDIR. If it's not specified, the system wide temporary directory TMPDIR will be used as the scratch directory.

1.3.4 Installation without network

If you have problems downloading the external libraries on your computer, you can manually build the libraries, as shown in the following instructions. First, you need to install libcint, libxc or xcfun libraries. libcint cint3 branch and xcfun stable-1.x branch are required by PySCF. They can be downloaded from github:

```
$ git clone https://github.com/sunqm/libcint.git
$ cd libcint
$ git checkout origin/cint3
$ cd .. && tar czf libcint.tar.gz libcint

$ git clone https://github.com/sunqm/xcfun.git
$ cd xcfun
$ git checkout origin/stable-1.x
$ cd .. && tar czf xcfun.tar.gz xcfun
```

libxc-3.* can be found in http://octopus-code.org/wiki/Main_Page or here. Assuming `/opt` is the place where these libraries will be installed, these packages should be compiled with the flags:

```
$ tar xvzf libcint.tar.gz
$ cd libcint
$ mkdir build && cd build
$ cmake -DWITH_F12=1 -DWITH_RANGE_COULOMB=1 -DWITH_COULOMB_ERF=1 \
    -DCMAKE_INSTALL_PREFIX:PATH=/opt -DCMAKE_INSTALL_LIBDIR:PATH=lib ..
$ make && make install

$ tar xvzf libxc-3.0.0.tar.gz
$ cd libxc-0.0.0
$ mkdir build && cd build
$ ../configure --prefix=/opt --libdir=/opt/lib --enable-shared --disable-fortran␣
↪LIBS=-lm
$ make && make install

$ tar xvzf xcfun.tar.gz
$ cd xcfun
$ mkdir build && cd build
$ cmake -DCMAKE_BUILD_TYPE=RELEASE -DBUILD_SHARED_LIBS=1 -DXC_MAX_ORDER=3 -DXCFUN_
↪ENABLE_TESTS=0 \
    -DCMAKE_INSTALL_PREFIX:PATH=/opt -DCMAKE_INSTALL_LIBDIR:PATH=lib ..
$ make && make install
```

Next compile PySCF:

```
$ cd pyscf/pyscf/lib
$ mkdir build && cd build
$ cmake -DBUILD_LIBCINT=0 -DBUILD_LIBXC=0 -DBUILD_XCFUN=0 -DCMAKE_INSTALL_PREFIX:
↪PATH=/opt ..
$ make
```

Finally update the `PYTHONPATH` environment for Python interpreter.

1.3.5 Using optimized BLAS

The default installation does not require the user to identify external linear algebra libraries, but instead tries to find them automatically. This automated setup script may only find and link to slow BLAS/LAPACK libraries. To improve performance, users can install the package with other BLAS vendors, such as the Intel Math Kernel Library (MKL), which can provide 10x speedup in many modules:

```
$ cd pyscf/lib/build
$ cmake -DBLA_VENDOR=Intel10_64lp_seq ..
$ make
```

If you are using Anaconda as your Python-side platform, you can link PySCF to the MKL library coming with Anaconda package:

```
$ export MKLROOT=/path/to/anaconda2
$ export LD_LIBRARY_PATH=$MKLROOT/lib:$LD_LIBRARY_PATH
$ cd pyscf/lib/build
$ cmake -DBLA_VENDOR=Intel10_64lp_seq ..
$ make
```

You can link to other BLAS libraries by setting `BLA_VENDOR`, eg `BLA_VENDOR=ATLAS`, `BLA_VENDOR=IBMESSL`. Please refer to cmake mannual for more details of the use of `FindBLAS` macro.

If the cmake `BLA_VENDOR` cannot find the right BLAS library as you expected, you can assign the libraries to the variable `BLAS_LIBRARIES` in `lib/CMakeLists.txt`:

```
set(BLAS_LIBRARIES "${BLAS_LIBRARIES};/path/to/mkl/lib/intel64/libmkl_intel_lp64.so")
set(BLAS_LIBRARIES "${BLAS_LIBRARIES};/path/to/mkl/lib/intel64/libmkl_sequential.so")
set(BLAS_LIBRARIES "${BLAS_LIBRARIES};/path/to/mkl/lib/intel64/libmkl_core.so")
set(BLAS_LIBRARIES "${BLAS_LIBRARIES};/path/to/mkl/lib/intel64/libmkl_avx.so")
```

1.3.6 Using optimized integral library

The default integral library used by PySCF is libcint (https://github.com/sunqm/libcint). To ensure the compatibility on various high performance computer systems, PySCF does not use the fast integral library by default. For X86-64 platforms, libcint library has an efficient implementation Qcint https://github.com/sunqm/qcint.git which is heavily optimized against SSE3 instructions. To replace the default libcint library with qcint library, edit the URL of the integral library in lib/CMakeLists.txt file:

```
ExternalProject_Add(libcint
   GIT_REPOSITORY
   https://github.com/sunqm/qcint.git
   ...
```

1.3.7 Plugins

nao

`pyscf/nao` module includes the basic functions of numerical atomic orbitals (NAO) and the (nao based) TDDFT methods. This module was contributed by Marc Barbry and Peter Koval. You can enable this module with a cmake flag:

```
$ cmake -DENABLE_NAO=1 ..
```

More information of the compilcation can be found in `pyscf/lib/nao/README.md`.

DMRG solver

Density matrix renormalization group (DMRG) implementations Block (http://chemists.princeton.edu/chan/software/block-code-for-dmrg) and CheMPS2 (http://sebwouters.github.io/CheMPS2/index.html) are efficient DMRG solvers

for ab initio quantum chemistry problem. Installing Block requires C++11 compiler. If C++11 is not supported by your compiler, you can register and download the precompiled Block binary from http://chemists.princeton.edu/chan/software/block-code-for-dmrg. Before using the Block or CheMPS2, you need create a config file future/dmrgscf/settings.py (as shown by settings.py.example) to store the path where the DMRG solver was installed.

FCIQMC

NECI (https://github.com/ghb24/NECI_STABLE) is FCIQMC code developed by George Booth and Ali Alavi. PySCF has an interface to call FCIQMC solver NECI. To use NECI, you need create a config file future/fciqmc/settings.py to store the path where NECI was installed.

Libxc

By default, building PySCF will automatically download and install Libxc 2.2.2. *pyscf.dft.libxc* module provided a general interface to access Libxc functionals.

Xcfun

By default, building PySCF will automatically download and install latest xcfun code from https://github.com/dftlibs/xcfun. `pyscf.dft.xcfun` module provided a general interface to access Libxc functionals.

XianCI

XianCI is a spin-adapted MRCI program. "Bingbing Suo" <bsuo@nwu.edu.cn> is the main developer of XianCI program.

1.4 gto — Molecular structure and GTO basis

This module provides the functions to parse the command line options, the molecular geometry and format the basic functions for *libcint* integral library. In `mole`, a basic class `Mole` is defined to hold the global parameters, which will be used throughout the package.

1.4.1 Input

Geometry

There are multiple ways to input molecular geometry. The internal format of `Mole.atom` is a python list:

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

You can input the geometry in this format. 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]])
```

Also, `atom` can be a string of Cartesian format or Z-matrix format:

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

There are a few requirements for the string format. The string input takes `;` or `\n` to partition atoms. White space and `,` are used to divide items for each atom. Blank lines or lines started with `#` will be ignored:

```
>>> mol = gto.M(
... mol.atom = '''
... #O 0 0 0
... H 0 1 0
...
... H 0 0 1;
... ''')
>>> mol.natm
2
```

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 rest, you can prefix or suffix number or special characters `1234567890~!@#$%^&*()_+.?:<>[]{}|` (except `,` and `;`) to an atomic symbol. With this decoration, you can specify different basis sets, or masses, or nuclear models for different atoms:

```
>>> mol = gto.Mole()
>>> mol.atom = '''8 0 0 0; h:1 0 1 0; H@2 0 0'''
>>> 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]]]
```

No matter which format or symbols were 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.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]]]
```

Input Basis

There are various ways to input basis sets. Besides the input of universal basis string and basis `dict`:

```
mol.basis = 'sto3g'
mol.basis = {'O': 'sto3g', 'H': '6-31g'}
```

basis can be input with helper functions. Function `basis.parse()` can parse a basis string of NWChem format (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
''')}
```

Functions `basis.load()` can be load arbitrary basis from the database, even the basis which does not match the element.

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

Both `basis.parse()` and `basis.load()` return the basis set in the internal format (See the basis_internal_format).

Basis parser supports "Ghost" atom:

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

More examples of inputing ghost atoms can be found in `examples/gto/03-ghost_atom.py`

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

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

Command line

Some of the input variables can be passed from command line:

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

This command line specifies the output file and the maximum of memory for the calculation. By default, command line has the highest priority, which means our 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 command line.

1.4.2 Program reference

mole

`Mole` class handles three layers: input, internal format, libcint arguments. The relationship of the three layers are:

```
.atom (input)  <=>  ._atom (for python) <=> ._atm (for libcint)
.basis (input) <=> ._basis (for python) <=> ._bas (for libcint)
```

input layer does not talk to libcint directly. Data are held in python internal fomrat layer. Most of methods defined in this class only operates on the internal format. Exceptions are make_env, make_atm_env, make_bas_env, `set_common_orig_()`, `set_rinv_orig_()` which are used to manipulate the libcint arguments.

pyscf.gto.mole.**M**(***kwargs*)

This is a shortcut to build up Mole object.

Args: Same to *Mole.build()*

Examples:

```
>>> from pyscf import gto
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='6-31g')
```

class pyscf.gto.mole.**Mole**(***kwargs*)

Basic class to hold molecular structure and global options

Attributes:

verbose [int] Print level

output [str or None] Output file, default is None which dumps msg to sys.stdout

max_memory [int, float] Allowed memory in MB

charge [int] Charge of molecule. It affects the electron numbers

spin [int] 2S, num. alpha electrons - num. beta electrons

symmetry [bool or str] Whether to use symmetry. When this variable is set to True, the molecule will be rotated and the highest rotation axis will be placed z-axis. If a string is given as the name of point group, the given point group symmetry will be used. Note that the input molecular coordinates will not be changed in this case.

symmetry_subgroup [str] subgroup

atom [list or str] To define molecluar structure. The internal format is

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

unit [str] Angstrom or Bohr

basis [dict or str] To define basis set.

nucmod [dict or str] Nuclear model. Set it to 0, None or False for point nuclear model. Any other values will enable Gaussian nuclear model. Default is point nuclear model.

cart [boolean] Using Cartesian GTO basis and integrals (6d,10f,15g)

** Following attributes are generated by *Mole.build()* **

stdout [file object] Default is sys.stdout if `Mole.output` is not set

groupname [str] One of D2h, C2h, C2v, D2, Cs, Ci, C2, C1

nelectron [int] sum of nuclear charges - `Mole.charge`

symm_orb [a list of numpy.ndarray] Symmetry adapted basis. Each element is a set of symm-adapted orbitals for one irreducible representation. The list index does **not** correspond to the id of irreducible representation.

irrep_id [a list of int] Each element is one irreducible representation id associated with the basis stored in symm_orb. One irrep id stands for one irreducible representation symbol. The irrep symbol and the relevant id are defined in `symm.param.IRREP_ID_TABLE`

irrep_name [a list of str] Each element is one irreducible representation symbol associated with the basis stored in symm_orb. The irrep symbols are defined in `symm.param.IRREP_ID_TABLE`

_built [bool] To label whether *Mole.build()* has been called. It ensures some functions being initialized once.

_basis [dict] like `Mole.basis`, the internal format which is returned from the parser *format_basis()*

_keys [a set of str] Store the keys appeared in the module. It is used to check misinput attributes

** Following attributes are arguments used by `libcint` library **

_atm : `[[charge,ptr-of-coord,nuc-model,ptr-zeta,0,0],[...]]` each element reperesents one atom

natm : number of atoms

_bas : `[[atom-id,angular-momentum,num-primitive-GTO,num-contracted-GTO,0,ptr-of-exps,` each element reperesents one shell

nbas : number of shells

_env : list of floats to store the coordinates, GTO exponents, contract-coefficients

Examples:

```
>>> mol = Mole(atom='H^2 0 0 0; H 0 0 1.1', basis='sto3g').build()
>>> print(mol.atom_symbol(0))
H^2
>>> print(mol.atom_pure_symbol(0))
H
>>> print(mol.nao_nr())
2
>>> print(mol.intor('int1e_ovlp_sph'))
[[ 0.99999999  0.43958641]
 [ 0.43958641  0.99999999]]
>>> mol.charge = 1
>>> mol.build()
<class 'pyscf.gto.mole.Mole'> has no attributes Charge
```

ao_labels(*mol*, *fmt=True*)

Labels for AO basis functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-AO-notation)] or formatted strings based on the argument "fmt"

ao_loc_2c(*mol*)
Offset of every shell in the spinor basis function spectrum

Returns: list, each entry is the corresponding start id of spinor function

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.ao_loc_2c(mol)
[0, 2, 4, 6, 12, 18, 20, 22, 24, 30, 36]
```

ao_loc_nr(*mol*, *cart=False*)
Offset of every shell in the spherical basis function spectrum

Returns: list, each entry is the corresponding start basis function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.ao_loc_nr(mol)
[0, 1, 2, 3, 6, 9, 10, 11, 12, 15, 18]
```

aoslice_2c_by_atom(*mol*)
2-component AO offset for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

aoslice_by_atom(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

aoslice_nr_by_atom(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

atom_charge(*atm_id*)
Nuclear effective charge of the given atom id Note "atom_charge /= _charge(atom_symbol)" when ECP is enabled. Number of electrons screened by ECP can be obtained by _charge(atom_symbol)-atom_charge

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_charge(1)
17
```

atom_charges()
np.asarray([mol.atom_charge(i) for i in range(mol.natm)])

atom_coord(*atm_id*)
Coordinates (ndarray) of the given atom id

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_coord(1)
[ 0.          0.          2.07869874]
```

atom_coords()
np.asarray([mol.atom_coords(i) for i in range(mol.natm)])

atom_nelec_core(*atm_id*)
Number of core electrons for pseudo potential.

atom_nshells(*atm_id*)
Number of basis/shells of the given atom

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_nshells(1)
5
```

atom_pure_symbol(*atm_id*)
For the given atom id, return the standard symbol (striping special characters)

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H^2 0 0 0; H 0 0 1.1')
>>> mol.atom_symbol(0)
H
```

atom_shell_ids(*atm_id*)
A list of the shell-ids of the given atom

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.atom_shell_ids(1)
[3, 4, 5, 6, 7]
```

atom_symbol(*atm_id*)
For the given atom id, return the input symbol (without striping special characters)

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H^2 0 0 0; H 0 0 1.1')
>>> mol.atom_symbol(0)
H^2
```

bas_angular(*bas_id*)
The angular momentum associated with the given basis

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(7)
2
```

bas_atom(*bas_id*)
The atom (0-based id) that the given basis sits on

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(7)
1
```

bas_coord(*bas_id*)
Coordinates (ndarray) associated with the given basis id

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.bas_coord(1)
[ 0.          0.          2.07869874]
```

bas_ctr_coeff(*bas_id*)
Contract coefficients (ndarray) of the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.M(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_ctr_coeff(0)
[[ 10.03400444]
 [  4.1188704 ]
 [  1.53971186]]
```

bas_exp(*bas_id*)
exponents (ndarray) of the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_kappa(0)
[ 13.01     1.962    0.4446]
```

bas_kappa(*bas_id*)
Kappa (if l < j, -l-1, else l) of the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_kappa(3)
0
```

bas_len_cart(*bas_id*)
The number of Cartesian function associated with given basis

bas_len_spinor(*bas_id*)
The number of spinor associated with given basis If kappa is 0, return 4l+2

bas_nctr(*bas_id*)
The number of contracted GTOs for the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(3)
3
```

bas_nprim(*bas_id*)
The number of primitive GTOs for the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(3)
11
```

build(*dump_input=True*, *parse_arg=True*, *verbose=None*, *output=None*, *max_memory=None*, *atom=None*, *basis=None*, *unit=None*, *nucmod=None*, *ecp=None*, *charge=None*, *spin=None*, *symmetry=None*, *symmetry_subgroup=None*, *cart=None*)
Setup moleclue and initialize some control parameters. Whenever you change the value of the attributes of *Mole*, you need call this function to refresh the internal data of Mole.

Kwargs:

dump_input [bool] whether to dump the contents of input file in the output file

parse_arg [bool] whether to read the sys.argv and overwrite the relevant parameters

verbose [int] Print level. If given, overwrite `Mole.verbose`

output [str or None] Output file. If given, overwrite `Mole.output`

max_memory [int, float] Allowd memory in MB. If given, overwrite `Mole.max_memory`

atom [list or str] To define molecluar structure.

basis [dict or str] To define basis set.

nucmod [dict or str] Nuclear model. If given, overwrite `Mole.nucmod`

charge [int] Charge of molecule. It affects the electron numbers If given, overwrite `Mole.charge`

spin [int] 2S, num. alpha electrons - num. beta electrons If given, overwrite `Mole.spin`

symmetry [bool or str] Whether to use symmetry. If given a string of point group name, the given point group symmetry will be used.

cart2sph_coeff (*normalized='sp'*)

Transformation matrix to transform the Cartesian GTOs to spherical GTOs

Kwargs:

normalized [string or boolean] How the Cartesian GTOs are normalized. Except s and p functions, Cartesian GTOs do not have the universal normalization coefficients for the different components of the same shell. The value of this argument can be one of 'sp', 'all', None. 'sp' means the Cartesian s and p basis are normalized. 'all' means all Cartesian functions are normalized. None means none of the Cartesian functions are normalized.

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvtz')
>>> c = mol.cart2sph_coeff()
>>> s0 = mol.intor('int1e_ovlp_sph')
>>> s1 = c.T.dot(mol.intor('int1e_ovlp_cart')).dot(c)
>>> print(abs(s1-s0).sum())
>>> 4.58676826646e-15
```

cart_labels (*fmt=False*)

Labels for Cartesian GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-xyz-notation)] or formatted strings based on the argument "fmt"

condense_to_shell (*mol*, *mat*, *compressor=<function amax>*)

The given matrix is first partitioned to blocks, based on AO shell as delimiter. Then call compressor function to abstract each block.

dumps (*mol*)

Serialize Mole object to a JSON formatted str.

energy_nuc (*mol*, *charges=None*, *coords=None*)

Compute nuclear repulsion energy (AU) or static Coulomb energy

Returns float

etbs (*etbs*)

Generate even tempered basis. See also *expand_etb()*

Args: etbs = [(l, n, alpha, beta), (l, n, alpha, beta),...]

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

eval_gto(*mol*, *eval_name*, *coords*, *comp=1*, *shls_slice=None*, *non0tab=None*, *ao_loc=None*, *out=None*)

Evaluate AO function value on the given grids,

Args: eval_name : str

Function	Expression
“GTOval_sph”	|AO>
“GTOval_ip_sph”	nabla |AO>
“GTOval_ig_sph”	(#C(0 1) g) |AO>
“GTOval_ipig_sph”	(#C(0 1) nabla g) |AO>
“GTOval_cart”	|AO>
“GTOval_ip_cart”	nabla |AO>
“GTOval_ig_cart”	(#C(0 1) g)|AO>

atm [int32 ndarray] libcint integral function argument

bas [int32 ndarray] libcint integral function argument

env [float64 ndarray] libcint integral function argument

coords [2D array, shape (N,3)] The coordinates of the grids.

Kwargs:

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.

non0tab [2D bool array] mask array to indicate whether the AO values are zero. The mask array can be obtained by calling `make_mask()`

out [ndarray] If provided, results are written into this array.

Returns: 2D array of shape (N,nao) Or 3D array of shape (*,N,nao) for AO values

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
>>> coords = numpy.random.random((100,3))  # 100 random points
>>> ao_value = mol.eval_gto("GTOval_sph", coords)
>>> print(ao_value.shape)
(100, 24)
>>> ao_value = mol.eval_gto("GTOval_ig_sph", coords, comp=3)
>>> print(ao_value.shape)
(3, 100, 24)
```

expand_etb(*l*, *n*, *alpha*, *beta*)

Generate the exponents of even tempered basis for `Mole.basis`. .. math:

```
e = e^{-\alpha * \beta^{i-1}} for i = 1 .. n
```

Args:

l [int] Angular momentum

n [int] Number of GTOs

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etb(1, 3, 1.5, 2)
[[1, [6.0, 1]], [1, [3.0, 1]], [1, [1.5, 1]]]
```

expand_etbs(*etbs*)

Generate even tempered basis. See also *expand_etb()*

Args: etbs = [(l, n, alpha, beta), (l, n, alpha, beta),...]

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

format_atom(*atom*, *origin=0*, *axes=None*, *unit='Ang'*)

Convert the input `Mole.atom` to the internal data format. Including, changing the nuclear charge to atom symbol, converting the coordinates to AU, rotate and shift the molecule. If the `atom` is a string, it takes ";" and "n" for the mark to separate atoms; "," and arbitrary length of blank space to spearate the individual terms for an atom. Blank line will be ignored.

Args:

atoms [list or str] the same to `Mole.atom`

Kwargs:

origin [ndarray] new axis origin.

axes [ndarray] (new_x, new_y, new_z), each entry is a length-3 array

unit [str or number] If unit is one of strings (B, b, Bohr, bohr, AU, au), the coordinates of the input atoms are the atomic unit; If unit is one of strings (A, a, Angstrom, angstrom, Ang, ang), the coordinates are in the unit of angstrom; If a number is given, the number are considered as the Bohr value (in angstrom), which should be around 0.53

Returns: "atoms" in the internal format as `_atom`

Examples:

```
>>> gto.format_atom('9,0,0,0; h@1 0 0 1', origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0]], ['H@1', [-1.0, -1.0, 0.0]]]
>>> gto.format_atom(['9,0,0,0', (1, (0, 0, 1))], origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0]], ['H', [-1, -1, 0]]]
```

format_basis(*basis_tab*)

Convert the input `Mole.basis` to the internal data format.

‘‘{ atom: [(l, ((-exp, c_1, c_2, ..),

(-exp, c_1, c_2, ..))),

(l, ((-exp, c_1, c_2, ..), (-exp, c_1, c_2, ..)))], ... }‘‘

Args:

basis_tab [dict] Similar to `Mole.basis`, it **cannot** be a str

Returns: Formated `basis`

Examples:

```
>>> gto.format_basis({'H':'sto-3g', 'H^2': '3-21g'})
{'H': [[0,
    [3.4252509099999999, 0.15432897000000001],
    [0.62391373000000006, 0.53532813999999995],
    [0.16885539999999999, 0.44463454000000002]]],
 'H^2': [[0,
    [5.4471780000000001, 0.15628500000000001],
    [0.82454700000000003, 0.90469100000000002]],
    [0, [0.18319199999999999, 1.0]]]}
```

format_ecp(*ecp_tab*)

Convert the input `ecp` (dict) to the internal data format:

```
{ atom: (nelec,  # core electrons
```

((l, # l=-1 for UL, l>=0 for Ul to indicate |l><l|

(((exp_1, c_1), # for r^0

(exp_2, c_2), ...),

((exp_1, c_1), # for r^1 (exp_2, c_2), ...),

((exp_1, c_1), # for r^2 ...))))),

...}

gto_norm(*l*, *expnt*)

Normalized factor for GTO radial part $g = r^l e^{-\alpha r^2}$

$$\frac{1}{\sqrt{\int g^2 r^2 dr}} = \sqrt{\frac{2^{2l+3}(l+1)!(2a)^{l+1.5}}{(2l+2)!\sqrt{\pi}}}$$

Ref: H. B. Schlegel and M. J. Frisch, Int. J. Quant. Chem., 54(1995), 83-87.

Args:

l (int): angular momentum

expnt : exponent α

Returns: normalization factor

Examples:

```
>>> print gto_norm(0, 1)
2.5264751109842591
```

has_ecp()

Whether pesudo potential is used in the system.

intor(*intor*, *comp=1*, *hermi=0*, *aosym='s1'*, *out=None*, *shls_slice=None*)

Integral generator.

Args:

intor [str] Name of the 1e or 2e AO integrals. Ref to `getints()` for the complete list of available 1-electron integral names

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp_sph has 3 components.

hermi [int] Symmetry of the integrals

0 : no symmetry assumed (default)
1 : hermitian
2 : anti-hermitian

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> mol.intor('int1e_ipnuc_sph', comp=3) # <nabla i | V_nuc | j>
[[[ 0.          0.        ]
  [ 0.          0.        ]]
 [[ 0.          0.        ]
  [ 0.          0.        ]]
 [[ 0.10289944  0.48176097]
  [-0.48176097 -0.10289944]]]
>>> mol.intor('int1e_nuc_spinor')
[[-1.69771092+0.j  0.00000000+0.j -0.67146312+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -1.69771092+0.j  0.00000000+0.j -0.67146312+0.j]
 [-0.67146312+0.j  0.00000000+0.j -1.69771092+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.67146312+0.j  0.00000000+0.j -1.69771092+0.j]]
```

intor_asymmetric(*intor*, *comp=1*)

One-electron integral generator. The integrals are assumed to be anti-hermitian

Args:

intor [str] Name of the 1-electron integral. Ref to `getints()` for the complete list of available 1-electron integral names

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp has 3 components.

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> mol.intor_asymmetric('int1e_nuc_spinor')
[[-1.69771092+0.j  0.00000000+0.j  0.67146312+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -1.69771092+0.j  0.00000000+0.j  0.67146312+0.j]
 [-0.67146312+0.j  0.00000000+0.j -1.69771092+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.67146312+0.j  0.00000000+0.j -1.69771092+0.j]]
```

intor_by_shell(*intor*, *shells*, *comp=1*)

For given 2, 3 or 4 shells, interface for libcint to get 1e, 2e, 2-center-2e or 3-center-2e integrals

Args:

intor_name [str] See also `getints()` for the supported intor_name

shls [list of int] The AO shell-ids of the integrals

atm [int32 ndarray] libcint integral function argument

bas [int32 ndarray] libcint integral function argument

env [float64 ndarray] libcint integral function argument

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp has 3 components.

Returns: ndarray of 2-dim to 5-dim, depending on the integral type (1e, 2e, 3c-2e, 2c2e) and the value of comp

Examples: The gradients of the spherical 2e integrals

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> gto.getints_by_shell('int2e_ip1_sph', (0,1,0,1), mol._atm, mol._bas, mol.
↪_env, comp=3)
[[[[-0.         ]]]]
  [[[[-0.         ]]]]
  [[[[-0.08760462]]]]]
```

intor_symmetric(*intor*, *comp=1*)

One-electron integral generator. The integrals are assumed to be hermitian

Args:

intor [str] Name of the 1-electron integral. Ref to `getints()` for the complete list of available 1-electron integral names

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp_sph has 3 components.

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> mol.intor_symmetric('int1e_nuc_spinor')
[[-1.69771092+0.j  0.00000000+0.j -0.67146312+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -1.69771092+0.j  0.00000000+0.j -0.67146312+0.j]
 [-0.67146312+0.j  0.00000000+0.j -1.69771092+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.67146312+0.j  0.00000000+0.j -1.69771092+0.j]]
```

kernel(*dump_input=True*, *parse_arg=True*, *verbose=None*, *output=None*, *max_memory=None*, *atom=None*, *basis=None*, *unit=None*, *nucmod=None*, *ecp=None*, *charge=None*, *spin=None*, *symmetry=None*, *symmetry_subgroup=None*, *cart=None*)

Setup moleclue and initialize some control parameters. Whenever you change the value of the attributes of *Mole*, you need call this function to refresh the internal data of Mole.

Kwargs:

dump_input [bool] whether to dump the contents of input file in the output file

parse_arg [bool] whether to read the sys.argv and overwrite the relevant parameters

verbose [int] Print level. If given, overwrite `Mole.verbose`

output [str or None] Output file. If given, overwrite `Mole.output`

max_memory [int, float] Allowd memory in MB. If given, overwrite `Mole.max_memory`

atom [list or str] To define molecluar structure.

basis [dict or str] To define basis set.

nucmod [dict or str] Nuclear model. If given, overwrite `Mole.nucmod`

charge [int] Charge of molecule. It affects the electron numbers If given, overwrite `Mole.charge`

spin [int] 2S, num. alpha electrons - num. beta electrons If given, overwrite `Mole.spin`

symmetry [bool or str] Whether to use symmetry. If given a string of point group name, the given point group symmetry will be used.

loads (*molstr*)
Deserialize a str containing a JSON document to a Mole object.

nao_2c (*mol*)
Total number of contracted spinor GTOs for the given *Mole* object

nao_2c_range (*mol*, *bas_id0*, *bas_id1*)
Lower and upper boundary of contracted spinor basis functions associated with the given shell range

Args:

mol : *Mole* object

bas_id0 [int] start shell id, 0-based

bas_id1 [int] stop shell id, 0-based

Returns: tupel of start basis function id and the stop function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.nao_2c_range(mol, 2, 4)
(4, 12)
```

nao_cart (*mol*)
Total number of contracted cartesian GTOs for the given *Mole* object

nao_nr (*mol*, *cart=False*)
Total number of contracted spherical GTOs for the given *Mole* object

nao_nr_range (*mol*, *bas_id0*, *bas_id1*)
Lower and upper boundary of contracted spherical basis functions associated with the given shell range

Args:

mol : *Mole* object

bas_id0 [int] start shell id

bas_id1 [int] stop shell id

Returns: tupel of start basis function id and the stop function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.nao_nr_range(mol, 2, 4)
(2, 6)
```

npgto_nr (*mol*, *cart=False*)
Total number of primitive spherical GTOs for the given *Mole* object

offset_2c_by_atom(*mol*)
2-component AO offset for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

offset_ao_by_atom(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

offset_nr_by_atom(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

pack(*mol*)
Pack the input args of *Mole* to a dict.

Note this function only pack the input arguments (not the entire Mole class). Modifications to mol._atm, mol._bas, mol._env are not tracked. Use *dumps()* to serialize the entire Mole object.

search_ao_label(*mol*, *label*)
Find the index of the AO basis function based on the given ao_label

Args:

ao_label [string or a list of strings] The regular expression pattern to match the orbital labels returned by mol.ao_labels()

Returns: A list of index for the AOs that matches the given ao_label RE pattern

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='ccpvtz')
>>> mol.parse_aolabel('Cl.*p')
[19 20 21 22 23 24 25 26 27 28 29 30]
>>> mol.parse_aolabel('Cl 2p')
[19 20 21]
>>> mol.parse_aolabel(['Cl.*d', 'Cl 4p'])
[25 26 27 31 32 33 34 35 36 37 38 39 40]
```

search_ao_nr(*mol*, *atm_id*, *l*, *m*, *atmshell*)
Search the first basis function id (**not** the shell id) which matches the given atom-id, angular momentum magnetic angular momentum, principal shell.

Args:

atm_id [int] atom id, 0-based

l [int] angular momentum

m [int] magnetic angular momentum

atmshell [int] principal quantum number

Returns: basis function id, 0-based. If not found, return None

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> mol.search_ao_nr(1, 1, -1, 3) # Cl 3px
7
```

set_common_orig(*coord*)
Update common origin which held in :class‘Mole‘._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_common_orig_(*coord*)
Update common origin which held in :class'Mole'._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_common_origin(*coord*)
Update common origin which held in :class'Mole'._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_common_origin_(*coord*)
Update common origin which held in :class'Mole'._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_f12_zeta(*zeta*)
Set zeta for YP exp(-zeta r12)/r12 or STG exp(-zeta r12) type integrals

set_geom_(*atoms*, *unit='Angstrom'*, *symmetry=None*)
Replace geometry

set_nuc_mod(*atm_id*, *zeta*)
Change the nuclear charge distribution of the given atom ID. The charge distribution is defined as: rho(r) = nuc_charge * Norm * exp(-zeta * r^2). This function can **only** be called after .build() method is executed.

Examples:

```
>>> for ia in range(mol.natm):
...     zeta = gto.filatov_nuc_mod(mol.atom_charge(ia))
...     mol.set_nuc_mod(ia, zeta)
```

set_nuc_mod_(*atm_id*, *zeta*)
Change the nuclear charge distribution of the given atom ID. The charge distribution is defined as: rho(r) = nuc_charge * Norm * exp(-zeta * r^2). This function can **only** be called after .build() method is executed.

Examples:

```
>>> for ia in range(mol.natm):
...     zeta = gto.filatov_nuc_mod(mol.atom_charge(ia))
...     mol.set_nuc_mod(ia, zeta)
```

set_range_coulomb(*omega*)
Apply the long range part of range-separated Coulomb operator for **all** 2e integrals erf(omega r12) / r12 set omega to 0 to siwtch off the range-separated Coulomb

set_range_coulomb_(*omega*)
Apply the long range part of range-separated Coulomb operator for **all** 2e integrals erf(omega r12) / r12 set omega to 0 to siwtch off the range-separated Coulomb

set_rinv_orig(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_orig_(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_origin(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_origin_(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_zeta(*zeta*)
Assume the charge distribution on the "rinv_orig". zeta is the parameter to control the charge distribution: rho(r) = Norm * exp(-zeta * r^2). **Be careful** when call this function. It affects the behavior of int1e_rinv_* functions. Make sure to set it back to 0 after using it!

set_rinv_zeta_(*zeta*)
Assume the charge distribution on the "rinv_orig". zeta is the parameter to control the charge distribution: rho(r) = Norm * exp(-zeta * r^2). **Be careful** when call this function. It affects the behavior of int1e_rinv_* functions. Make sure to set it back to 0 after using it!

spheric_labels(*fmt=False*)
Labels for spherical GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-real-spherical-notation] or formatted strings based on the argument "fmt"

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.spherical_labels(mol)
[(0, 'H', '1s', ''), (1, 'Cl', '1s', ''), (1, 'Cl', '2s', ''), (1, 'Cl', '3s
↪', ''), (1, 'Cl', '2p', 'x'), (1, 'Cl', '2p', 'y'), (1, 'Cl', '2p', 'z'),␣
↪(1, 'Cl', '3p', 'x'), (1, 'Cl', '3p', 'y'), (1, 'Cl', '3p', 'z')]
```

spherical_labels(*fmt=False*)
Labels for spherical GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-real-spherical-notation] or formatted strings based on the argument "fmt"

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.spherical_labels(mol)
[(0, 'H', '1s', ''), (1, 'Cl', '1s', ''), (1, 'Cl', '2s', ''), (1, 'Cl', '3s
↪', ''), (1, 'Cl', '2p', 'x'), (1, 'Cl', '2p', 'y'), (1, 'Cl', '2p', 'z'),␣
↪(1, 'Cl', '3p', 'x'), (1, 'Cl', '3p', 'y'), (1, 'Cl', '3p', 'z')]
```

time_reversal_map(*mol*)

The index to map the spinor functions and its time reversal counterpart. The returned indices have postive or negative values. For the i-th basis function, if the returned j = idx[i] < 0, it means $T|i\rangle = -|j\rangle$, otherwise $T|i\rangle = |j\rangle$

tot_electrons(*mol*)

Total number of electrons for the given molecule

Returns: electron number in integer

Examples:

```
>>> mol = gto.M(atom='H 0 1 0; C 0 0 1', charge=1)
>>> mol.tot_electrons()
6
```

unpack(*moldic*)

Unpack a dict which is packed by *pack()*, to generate the input arguments for *Mole* object.

pyscf.gto.mole.**ao_labels**(*mol*, *fmt=True*)

Labels for AO basis functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-AO-notation)] or formatted strings based on the argument "fmt"

pyscf.gto.mole.**ao_loc_2c**(*mol*)

Offset of every shell in the spinor basis function spectrum

Returns: list, each entry is the corresponding start id of spinor function

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.ao_loc_2c(mol)
[0, 2, 4, 6, 12, 18, 20, 22, 24, 30, 36]
```

pyscf.gto.mole.**ao_loc_nr**(*mol*, *cart=False*)

Offset of every shell in the spherical basis function spectrum

Returns: list, each entry is the corresponding start basis function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.ao_loc_nr(mol)
[0, 1, 2, 3, 6, 9, 10, 11, 12, 15, 18]
```

pyscf.gto.mole.**aoslice_by_atom**(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

pyscf.gto.mole.**atom_types**(*atoms*, *basis=None*)
symmetry inequivalent atoms

pyscf.gto.mole.**cart2j_kappa**(*kappa*, *l=None*, *normalized=None*)
Cartesian to spinor, indexed by kappa

Kwargs:

normalized : How the Cartesian GTOs are normalized. 'sp' means the s and p functions are normalized.

pyscf.gto.mole.**cart2j_l**(*l*, *normalized=None*)
Cartesian to spinor, indexed by l

pyscf.gto.mole.**cart2sph**(*l*)
Cartesian to real spherical transformation matrix

pyscf.gto.mole.**cart2zmat**(*coord*)

```
>>> c = numpy.array((
(0.000000000000,  1.889726124565,  0.000000000000),
(0.000000000000,  0.000000000000, -1.889726124565),
(1.889726124565, -1.889726124565,  0.000000000000),
(1.889726124565,  0.000000000000,  1.133835674739)))
>>> print cart2zmat(c)
1
1 2.67247631453057
1 4.22555607338457 2 50.7684795164077
1 2.90305235726773 2 79.3904651036893 3 6.20854462618583
```

pyscf.gto.mole.**cart_labels**(*mol*, *fmt=True*)
Labels for Cartesian GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-xyz-notation)] or formatted strings based on the argument "fmt"

pyscf.gto.mole.**chiral_mol**(*mol1*, *mol2=None*)
Detect whether the given molelcule is chiral molecule or two molecules are chiral isomers.

pyscf.gto.mole.**conc_env**(*atm1*, *bas1*, *env1*, *atm2*, *bas2*, *env2*)
Concatenate two Mole's integral parameters. This function can be used to construct the environment for cross integrals like

$$\langle\mu|\nu\rangle, \mu \in mol1, \nu \in mol2$$

Returns: Concatenated atm, bas, env

Examples: Compute the overlap between H2 molecule and O atom

```
>>> mol1 = gto.M(atom='H 0 1 0; H 0 0 1', basis='sto3g')
>>> mol2 = gto.M(atom='O 0 0 0', basis='sto3g')
>>> atm3, bas3, env3 = gto.conc_env(mol1._atm, mol1._bas, mol1._env,
...                                 mol2._atm, mol2._bas, mol2._env)
>>> gto.moleintor.getints('int1e_ovlp_sph', atm3, bas3, env3, range(2),␣
↪range(2,5))
[[ 0.04875181  0.44714688  0.          0.37820346  0.        ]
 [ 0.04875181  0.44714688  0.          0.          0.37820346]]
```

pyscf.gto.mole.**conc_mol**(*mol1*, *mol2*)
Concatenate two Mole objects.

pyscf.gto.mole.**condense_to_shell**(*mol*, *mat*, *compressor=<function amax>*)
The given matrix is first partitioned to blocks, based on AO shell as delimiter. Then call compressor function to abstract each block.

pyscf.gto.mole.**copy**(*mol*)
Deepcopy of the given *Mole* object

pyscf.gto.mole.**dumps**(*mol*)
Serialize Mole object to a JSON formatted str.

pyscf.gto.mole.**dyall_nuc_mod**(*mass*, *c=137.03599967994*)
Generate the nuclear charge distribution parameter zeta rho(r) = nuc_charge * Norm * exp(-zeta * r^2)

Ref. L. Visscher and K. Dyall, At. Data Nucl. Data Tables, 67, 207 (1997)

pyscf.gto.mole.**energy_nuc**(*mol*, *charges=None*, *coords=None*)
Compute nuclear repulsion energy (AU) or static Coulomb energy

Returns float

pyscf.gto.mole.**etbs**(*etbs*)
Generate even tempered basis. See also *expand_etb()*

Args: etbs = [(l, n, alpha, beta), (l, n, alpha, beta),...]

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

pyscf.gto.mole.**expand_etb**(*l*, *n*, *alpha*, *beta*)
Generate the exponents of even tempered basis for `Mole.basis`. .. math:

```
e = e^{-\alpha * \beta^{i-1}} for i = 1 .. n
```

Args:

l [int] Angular momentum

n [int] Number of GTOs

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etb(1, 3, 1.5, 2)
[[1, [6.0, 1]], [1, [3.0, 1]], [1, [1.5, 1]]]
```

pyscf.gto.mole.**expand_etbs**(*etbs*)
Generate even tempered basis. See also *expand_etb()*

Args: etbs = [(l, n, alpha, beta), (l, n, alpha, beta),...]

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

pyscf.gto.mole.**filatov_nuc_mod**(*nuc_charge*, *c=137.03599967994*)
Generate the nuclear charge distribution parameter zeta rho(r) = nuc_charge * Norm * exp(-zeta * r^2)

Ref. M. Filatov and D. Cremer, Theor. Chem. Acc. 108, 168 (2002)

13. Filatov and D. Cremer, Chem. Phys. Lett. 351, 259 (2002)

pyscf.gto.mole.**format_atom**(*atoms*, *origin=0*, *axes=None*, *unit='Ang'*)
Convert the input `Mole.atom` to the internal data format. Including, changing the nuclear charge to atom symbol, converting the coordinates to AU, rotate and shift the molecule. If the `atom` is a string, it takes ";" and "n" for the mark to separate atoms; "," and arbitrary length of blank space to spearate the individual terms for an atom. Blank line will be ignored.

Args:

atoms [list or str] the same to `Mole.atom`

Kwargs:

origin [ndarray] new axis origin.

axes [ndarray] (new_x, new_y, new_z), each entry is a length-3 array

unit [str or number] If unit is one of strings (B, b, Bohr, bohr, AU, au), the coordinates of the input atoms are the atomic unit; If unit is one of strings (A, a, Angstrom, angstrom, Ang, ang), the coordinates are in the unit of angstrom; If a number is given, the number are considered as the Bohr value (in angstrom), which should be around 0.53

Returns: "atoms" in the internal format as `_atom`

Examples:

```
>>> gto.format_atom('9,0,0,0; h@1 0 0 1', origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0]], ['H@1', [-1.0, -1.0, 0.0]]]
>>> gto.format_atom(['9,0,0,0', (1, (0, 0, 1))], origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0]], ['H', [-1, -1, 0]]]
```

pyscf.gto.mole.**format_basis**(*basis_tab*)
Convert the input `Mole.basis` to the internal data format.

‘‘**{ atom: [(l, ((-exp, c_1, c_2, ..),**

(-exp, c_1, c_2, ..))),

(l, ((-exp, c_1, c_2, ..), (-exp, c_1, c_2, ..)))], ... }‘‘

Args:

basis_tab [dict] Similar to `Mole.basis`, it **cannot** be a str

Returns: Formated `basis`

Examples:

```
>>> gto.format_basis({'H':'sto-3g', 'H^2': '3-21g'})
{'H': [[0,
    [3.4252509099999999, 0.15432897000000001],
    [0.62391373000000006, 0.53532813999999995],
    [0.16885539999999999, 0.44463454000000002]]],
 'H^2': [[0,
    [5.4471780000000001, 0.15628500000000001],
    [0.82454700000000003, 0.90469100000000002]],
    [0, [0.18319199999999999, 1.0]]]}
```

pyscf.gto.mole.**format_ecp**(*ecp_tab*)

Convert the input `ecp` (dict) to the internal data format:

```
{ atom: (nelec,  # core electrons
```

((l, # l=-1 for UL, l>=0 for Ul to indicate |l><l|

(((exp_1, c_1), # for r^0

(exp_2, c_2), ...),

((exp_1, c_1), # for r^1 (exp_2, c_2), ...),

((exp_1, c_1), # for r^2 ...))))),

...}

pyscf.gto.mole.**from_zmatrix**(*atomstr*)

```
>>> a = """H
H 1 2.67247631453057
H 1 4.22555607338457 2 50.7684795164077
H 1 2.90305235726773 2 79.3904651036893 3 6.20854462618583"""
>>> for x in zmat2cart(a): print x
['H', array([ 0.,  0.,  0.])]
['H', array([ 2.67247631,  0.        ,  0.        ])]
['H', array([ 2.67247631,  0.        ,  3.27310166])]
['H', array([ 0.53449526,  0.30859098,  2.83668811])]
```

pyscf.gto.mole.**gto_norm**(*l*, *expnt*)

Normalized factor for GTO radial part $g = r^l e^{-\alpha r^2}$

$$\frac{1}{\sqrt{\int g^2 r^2 dr}} = \sqrt{\frac{2^{2l+3}(l+1)!(2a)^{l+1.5}}{(2l+2)!\sqrt{\pi}}}$$

Ref: H. B. Schlegel and M. J. Frisch, Int. J. Quant. Chem., 54(1995), 83-87.

Args:

l (int): angular momentum

expnt : exponent α

Returns: normalization factor

Examples:

```
>>> print gto_norm(0, 1)
2.5264751109842591
```

pyscf.gto.mole.**intor_cross**(*intor*, *mol1*, *mol2*, *comp=1*)
1-electron integrals from two molecules like

$$\langle\mu|intor|\nu\rangle, \mu \in mol1, \nu \in mol2$$

Args:

intor [str] Name of the 1-electron integral, such as int1e_ovlp_sph (spherical overlap), int1e_nuc_cart (cartesian nuclear attraction), int1e_ipovlp_spinor (spinor overlap gradients), etc. Ref to `getints()` for the full list of available 1-electron integral names

mol1, mol2: *Mole* objects

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp_sph has 3 components

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples: Compute the overlap between H2 molecule and O atom

```
>>> mol1 = gto.M(atom='H 0 1 0; H 0 0 1', basis='sto3g')
>>> mol2 = gto.M(atom='O 0 0 0', basis='sto3g')
>>> gto.intor_cross('int1e_ovlp_sph', mol1, mol2)
[[ 0.04875181  0.44714688  0.          0.37820346  0.        ]
 [ 0.04875181  0.44714688  0.          0.          0.37820346]]
```

pyscf.gto.mole.**is_same_mol**(*mol1*, *mol2*, *tol=1e-05*, *cmp_basis=True*, *ignore_chiral=False*)
Compare the two molecules whether they have the same structure.

Kwargs:

tol [float] In Bohr

cmp_basis [bool] Whether to compare basis functions for the two molecules

pyscf.gto.mole.**len_cart**(*l*)
The number of Cartesian function associated with given angular momentum.

pyscf.gto.mole.**len_spinor**(*l*, *kappa*)
The number of spinor associated with given angular momentum and kappa. If kappa is 0, return 4l+2

pyscf.gto.mole.**loads**(*molstr*)
Deserialize a str containing a JSON document to a Mole object.

pyscf.gto.mole.**make_atm_env**(*atom*, *ptr=0*)
Convert the internal format `Mole._atom` to the format required by `libcint` integrals

pyscf.gto.mole.**make_bas_env**(*basis_add*, *atom_id=0*, *ptr=0*)
Convert `Mole.basis` to the argument `bas` for `libcint` integrals

pyscf.gto.mole.**make_env**(*atoms*, *basis*, *pre_env=[]*, *nucmod={}*)
Generate the input arguments for `libcint` library based on internal format `Mole._atom` and `Mole._basis`

pyscf.gto.mole.**nao_2c**(*mol*)
Total number of contracted spinor GTOs for the given *Mole* object

pyscf.gto.mole.**nao_2c_range**(*mol*, *bas_id0*, *bas_id1*)
Lower and upper boundary of contracted spinor basis functions associated with the given shell range

Args:

mol : *Mole* object

bas_id0 [int] start shell id, 0-based

bas_id1 [int] stop shell id, 0-based

Returns: tupel of start basis function id and the stop function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.nao_2c_range(mol, 2, 4)
(4, 12)
```

pyscf.gto.mole.**nao_cart**(*mol*)
Total number of contracted cartesian GTOs for the given *Mole* object

pyscf.gto.mole.**nao_nr**(*mol*, *cart=False*)
Total number of contracted spherical GTOs for the given *Mole* object

pyscf.gto.mole.**nao_nr_range**(*mol*, *bas_id0*, *bas_id1*)
Lower and upper boundary of contracted spherical basis functions associated with the given shell range

Args:

mol : *Mole* object

bas_id0 [int] start shell id

bas_id1 [int] stop shell id

Returns: tupel of start basis function id and the stop function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.nao_nr_range(mol, 2, 4)
(2, 6)
```

pyscf.gto.mole.**npgto_nr**(*mol*, *cart=False*)
Total number of primitive spherical GTOs for the given *Mole* object

pyscf.gto.mole.**offset_2c_by_atom**(*mol*)
2-component AO offset for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

pyscf.gto.mole.**offset_nr_by_atom**(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

pyscf.gto.mole.**pack**(*mol*)
Pack the input args of *Mole* to a dict.

Note this function only pack the input arguments (not the entire Mole class). Modifications to mol._atm, mol._bas, mol._env are not tracked. Use *dumps()* to serialize the entire Mole object.

pyscf.gto.mole.**same_mol**(*mol1*, *mol2*, *tol=1e-05*, *cmp_basis=True*, *ignore_chiral=False*)
Compare the two molecules whether they have the same structure.

Kwargs:

tol [float] In Bohr

cmp_basis [bool] Whether to compare basis functions for the two molecules

pyscf.gto.mole.**search_ao_label**(*mol*, *label*)

Find the index of the AO basis function based on the given ao_label

Args:

ao_label [string or a list of strings] The regular expression pattern to match the orbital labels returned by mol.ao_labels()

Returns: A list of index for the AOs that matches the given ao_label RE pattern

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='ccpvtz')
>>> mol.parse_aolabel('Cl.*p')
[19 20 21 22 23 24 25 26 27 28 29 30]
>>> mol.parse_aolabel('Cl 2p')
[19 20 21]
>>> mol.parse_aolabel(['Cl.*d', 'Cl 4p'])
[25 26 27 31 32 33 34 35 36 37 38 39 40]
```

pyscf.gto.mole.**search_ao_nr**(*mol*, *atm_id*, *l*, *m*, *atmshell*)

Search the first basis function id (**not** the shell id) which matches the given atom-id, angular momentum magnetic angular momentum, principal shell.

Args:

atm_id [int] atom id, 0-based

l [int] angular momentum

m [int] magnetic angular momentum

atmshell [int] principal quantum number

Returns: basis function id, 0-based. If not found, return None

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> mol.search_ao_nr(1, 1, -1, 3) # Cl 3px
7
```

pyscf.gto.mole.**search_shell_id**(*mol*, *atm_id*, *l*)

Search the first basis/shell id (**not** the basis function id) which matches the given atom-id and angular momentum

Args:

atm_id [int] atom id, 0-based

l [int] angular momentum

Returns: basis id, 0-based. If not found, return None

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> mol.search_shell_id(1, 1) # Cl p shell
4
>>> mol.search_shell_id(1, 2) # Cl d shell
None
```

pyscf.gto.mole.**spheric_labels**(*mol*, *fmt=True*)
Labels for spherical GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-real-spherical-notation] or formatted strings based on the argument "fmt"

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.spherical_labels(mol)
[(0, 'H', '1s', ''), (1, 'Cl', '1s', ''), (1, 'Cl', '2s', ''), (1, 'Cl', '3s', '
↪'), (1, 'Cl', '2p', 'x'), (1, 'Cl', '2p', 'y'), (1, 'Cl', '2p', 'z'), (1, 'Cl',
↪'3p', 'x'), (1, 'Cl', '3p', 'y'), (1, 'Cl', '3p', 'z')]
```

pyscf.gto.mole.**spherical_labels**(*mol*, *fmt=True*)
Labels for spherical GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-real-spherical-notation] or formatted strings based on the argument "fmt"

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.spherical_labels(mol)
[(0, 'H', '1s', ''), (1, 'Cl', '1s', ''), (1, 'Cl', '2s', ''), (1, 'Cl', '3s', '
↪'), (1, 'Cl', '2p', 'x'), (1, 'Cl', '2p', 'y'), (1, 'Cl', '2p', 'z'), (1, 'Cl',
↪'3p', 'x'), (1, 'Cl', '3p', 'y'), (1, 'Cl', '3p', 'z')]
```

pyscf.gto.mole.**time_reversal_map**(*mol*)
The index to map the spinor functions and its time reversal counterpart. The returned indices have postive or negative values. For the i-th basis function, if the returned j = idx[i] < 0, it means $T|i\rangle = -|j\rangle$, otherwise $T|i\rangle = |j\rangle$

pyscf.gto.mole.**tot_electrons**(*mol*)
Total number of electrons for the given molecule

Returns: electron number in integer

Examples:

```
>>> mol = gto.M(atom='H 0 1 0; C 0 0 1', charge=1)
>>> mol.tot_electrons()
6
```

pyscf.gto.mole.**uncontract**(*_basis*)
Uncontract internal format _basis

Examples:

```
>>> gto.uncontract_basis(gto.load('sto3g', 'He'))
[[0, [6.3624213899999997, 1]], [0, [1.1589229999999999, 1]], [0, [0.
↪31364978999999998, 1]]]
```

pyscf.gto.mole.**uncontract_basis**(*_basis*)
Uncontract internal format _basis

Examples:

```
>>> gto.uncontract_basis(gto.load('sto3g', 'He'))
[[0, [6.3624213899999997, 1]], [0, [1.1589229999999999, 1]], [0, [0.
↪31364978999999998, 1]]]
```

pyscf.gto.mole.**unpack**(*moldic*)

Unpack a dict which is packed by *pack()*, to generate the input arguments for *Mole* object.

pyscf.gto.mole.**zmat**(*atomstr*)

```
>>> a = """H
H 1 2.67247631453057
H 1 4.22555607338457 2 50.7684795164077
H 1 2.90305235726773 2 79.3904651036893 3 6.20854462618583"""
>>> for x in zmat2cart(a): print x
['H', array([ 0.,  0.,  0.])]
['H', array([ 2.67247631,  0.        ,  0.        ])]
['H', array([ 2.67247631,  0.        ,  3.27310166])]
['H', array([ 0.53449526,  0.30859098,  2.83668811])]
```

pyscf.gto.mole.**zmat2cart**(*atomstr*)

```
>>> a = """H
H 1 2.67247631453057
H 1 4.22555607338457 2 50.7684795164077
H 1 2.90305235726773 2 79.3904651036893 3 6.20854462618583"""
>>> for x in zmat2cart(a): print x
['H', array([ 0.,  0.,  0.])]
['H', array([ 2.67247631,  0.        ,  0.        ])]
['H', array([ 2.67247631,  0.        ,  3.27310166])]
['H', array([ 0.53449526,  0.30859098,  2.83668811])]
```

class pyscf.gto.mole.**Mole**(***kwargs*)

Basic class to hold molecular structure and global options

Attributes:

verbose [int] Print level

output [str or None] Output file, default is None which dumps msg to sys.stdout

max_memory [int, float] Allowed memory in MB

charge [int] Charge of molecule. It affects the electron numbers

spin [int] 2S, num. alpha electrons - num. beta electrons

symmetry [bool or str] Whether to use symmetry. When this variable is set to True, the molecule will be rotated and the highest rotation axis will be placed z-axis. If a string is given as the name of point group, the given point group symmetry will be used. Note that the input molecular coordinates will not be changed in this case.

symmetry_subgroup [str] subgroup

atom [list or str] To define molecluar structure. The internal format is

atom = [[atom1, (x, y, z)],

[atom2, (x, y, z)],
...
[atomN, (x, y, z)]]

unit [str] Angstrom or Bohr

basis [dict or str] To define basis set.

nucmod [dict or str] Nuclear model. Set it to 0, None or False for point nuclear model. Any other values will enable Gaussian nuclear model. Default is point nuclear model.

cart [boolean] Using Cartesian GTO basis and integrals (6d,10f,15g)

** Following attributes are generated by *Mole.build()* **

stdout [file object] Default is sys.stdout if `Mole.output` is not set

groupname [str] One of D2h, C2h, C2v, D2, Cs, Ci, C2, C1

nelectron [int] sum of nuclear charges - `Mole.charge`

symm_orb [a list of numpy.ndarray] Symmetry adapted basis. Each element is a set of symm-adapted orbitals for one irreducible representation. The list index does **not** correspond to the id of irreducible representation.

irrep_id [a list of int] Each element is one irreducible representation id associated with the basis stored in symm_orb. One irrep id stands for one irreducible representation symbol. The irrep symbol and the relevant id are defined in `symm.param.IRREP_ID_TABLE`

irrep_name [a list of str] Each element is one irreducible representation symbol associated with the basis stored in symm_orb. The irrep symbols are defined in `symm.param.IRREP_ID_TABLE`

_built [bool] To label whether *Mole.build()* has been called. It ensures some functions being initialized once.

_basis [dict] like `Mole.basis`, the internal format which is returned from the parser *format_basis()*

_keys [a set of str] Store the keys appeared in the module. It is used to check misinput attributes

** Following attributes are arguments used by `libcint` library **

_atm : `[[charge,ptr-of-coord,nuc-model,ptr-zeta,0,0],[...]]` each element reperesents one atom

natm : number of atoms

_bas : `[[atom-id,angular-momentum,num-primitive-GTO,num-contracted-GTO,0,ptr-of-exps` each element reperesents one shell

nbas : number of shells

_env : list of floats to store the coordinates, GTO exponents, contract-coefficients

Examples:

```
>>> mol = Mole(atom='H^2 0 0 0; H 0 0 1.1', basis='sto3g').build()
>>> print(mol.atom_symbol(0))
H^2
>>> print(mol.atom_pure_symbol(0))
H
>>> print(mol.nao_nr())
2
```

```
>>> print(mol.intor('int1e_ovlp_sph'))
[[ 0.99999999  0.43958641]
 [ 0.43958641  0.99999999]]
>>> mol.charge = 1
>>> mol.build()
<class 'pyscf.gto.mole.Mole'> has no attributes Charge
```

ao_labels(*mol*, *fmt=True*)

Labels for AO basis functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-AO-notation)] or formatted strings based on the argument "fmt"

ao_loc_2c(*mol*)

Offset of every shell in the spinor basis function spectrum

Returns: list, each entry is the corresponding start id of spinor function

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.ao_loc_2c(mol)
[0, 2, 4, 6, 12, 18, 20, 22, 24, 30, 36]
```

ao_loc_nr(*mol*, *cart=False*)

Offset of every shell in the spherical basis function spectrum

Returns: list, each entry is the corresponding start basis function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.ao_loc_nr(mol)
[0, 1, 2, 3, 6, 9, 10, 11, 12, 15, 18]
```

aoslice_2c_by_atom(*mol*)

2-component AO offset for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

aoslice_by_atom(*mol*, *ao_loc=None*)

AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

aoslice_nr_by_atom(*mol*, *ao_loc=None*)

AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

atom_charge(*atm_id*)

Nuclear effective charge of the given atom id Note "atom_charge /= _charge(atom_symbol)" when ECP is enabled. Number of electrons screened by ECP can be obtained by _charge(atom_symbol)-atom_charge

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_charge(1)
17
```

atom_charges()
np.asarray([mol.atom_charge(i) for i in range(mol.natm)])

atom_coord(*atm_id*)
Coordinates (ndarray) of the given atom id

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_coord(1)
[ 0.          0.          2.07869874]
```

atom_coords()
np.asarray([mol.atom_coords(i) for i in range(mol.natm)])

atom_nelec_core(*atm_id*)
Number of core electrons for pseudo potential.

atom_nshells(*atm_id*)
Number of basis/shells of the given atom

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.atom_nshells(1)
5
```

atom_pure_symbol(*atm_id*)
For the given atom id, return the standard symbol (striping special characters)

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H^2 0 0 0; H 0 0 1.1')
>>> mol.atom_symbol(0)
H
```

atom_shell_ids(*atm_id*)
A list of the shell-ids of the given atom

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.atom_shell_ids(1)
[3, 4, 5, 6, 7]
```

atom_symbol(*atm_id*)

For the given atom id, return the input symbol (without striping special characters)

Args:

atm_id [int] 0-based

Examples:

```
>>> mol.build(atom='H^2 0 0 0; H 0 0 1.1')
>>> mol.atom_symbol(0)
H^2
```

bas_angular(*bas_id*)

The angular momentum associated with the given basis

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(7)
2
```

bas_atom(*bas_id*)

The atom (0-based id) that the given basis sits on

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(7)
1
```

bas_coord(*bas_id*)

Coordinates (ndarray) associated with the given basis id

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1')
>>> mol.bas_coord(1)
[ 0.          0.          2.07869874]
```

bas_ctr_coeff(*bas_id*)

Contract coefficients (ndarray) of the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.M(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_ctr_coeff(0)
[[ 10.03400444]
 [  4.1188704 ]
 [  1.53971186]]
```

bas_exp(*bas_id*)
exponents (ndarray) of the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_kappa(0)
[ 13.01     1.962    0.4446]
```

bas_kappa(*bas_id*)
Kappa (if l < j, -l-1, else l) of the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_kappa(3)
0
```

bas_len_cart(*bas_id*)
The number of Cartesian function associated with given basis

bas_len_spinor(*bas_id*)
The number of spinor associated with given basis If kappa is 0, return 4l+2

bas_nctr(*bas_id*)
The number of contracted GTOs for the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(3)
3
```

bas_nprim(*bas_id*)
The number of primitive GTOs for the given shell

Args:

bas_id [int] 0-based

Examples:

```
>>> mol.build(atom='H 0 0 0; Cl 0 0 1.1', basis='cc-pvdz')
>>> mol.bas_atom(3)
11
```

build(*dump_input=True*, *parse_arg=True*, *verbose=None*, *output=None*, *max_memory=None*, *atom=None*, *basis=None*, *unit=None*, *nucmod=None*, *ecp=None*, *charge=None*, *spin=None*, *symmetry=None*, *symmetry_subgroup=None*, *cart=None*)

Setup moleclue and initialize some control parameters. Whenever you change the value of the attributes of *Mole*, you need call this function to refresh the internal data of Mole.

Kwargs:

dump_input [bool] whether to dump the contents of input file in the output file

parse_arg [bool] whether to read the sys.argv and overwrite the relevant parameters

verbose [int] Print level. If given, overwrite `Mole.verbose`

output [str or None] Output file. If given, overwrite `Mole.output`

max_memory [int, float] Allowd memory in MB. If given, overwrite `Mole.max_memory`

atom [list or str] To define molecluar structure.

basis [dict or str] To define basis set.

nucmod [dict or str] Nuclear model. If given, overwrite `Mole.nucmod`

charge [int] Charge of molecule. It affects the electron numbers If given, overwrite `Mole.charge`

spin [int] 2S, num. alpha electrons - num. beta electrons If given, overwrite `Mole.spin`

symmetry [bool or str] Whether to use symmetry. If given a string of point group name, the given point group symmetry will be used.

cart2sph_coeff(*normalized='sp'*)

Transformation matrix to transform the Cartesian GTOs to spherical GTOs

Kwargs:

normalized [string or boolean] How the Cartesian GTOs are normalized. Except s and p functions, Cartesian GTOs do not have the universal normalization coefficients for the different components of the same shell. The value of this argument can be one of 'sp', 'all', None. 'sp' means the Cartesian s and p basis are normalized. 'all' means all Cartesian functions are normalized. None means none of the Cartesian functions are normalized.

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvtz')
>>> c = mol.cart2sph_coeff()
>>> s0 = mol.intor('int1e_ovlp_sph')
>>> s1 = c.T.dot(mol.intor('int1e_ovlp_cart')).dot(c)
>>> print(abs(s1-s0).sum())
>>> 4.58676826646e-15
```

cart_labels(*fmt=False*)

Labels for Cartesian GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-xyz-notation)] or formatted strings based on the argument "fmt"

condense_to_shell (*mol*, *mat*, *compressor=<function amax>*)
The given matrix is first partitioned to blocks, based on AO shell as delimiter. Then call compressor function to abstract each block.

dumps (*mol*)
Serialize Mole object to a JSON formatted str.

energy_nuc (*mol*, *charges=None*, *coords=None*)
Compute nuclear repulsion energy (AU) or static Coulomb energy

Returns float

etbs (*etbs*)
Generate even tempered basis. See also *expand_etb()*

Args: etbs = [(l, n, alpha, beta), (l, n, alpha, beta),...]

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

eval_gto (*mol*, *eval_name*, *coords*, *comp=1*, *shls_slice=None*, *non0tab=None*, *ao_loc=None*, *out=None*)
Evaluate AO function value on the given grids,

Args: eval_name : str

Function	Expression
“GTOval_sph”	|AO>
“GTOval_ip_sph”	nabla |AO>
“GTOval_ig_sph”	(#C(0 1) g) |AO>
“GTOval_ipig_sph”	(#C(0 1) nabla g) |AO>
“GTOval_cart”	|AO>
“GTOval_ip_cart”	nabla |AO>
“GTOval_ig_cart”	(#C(0 1) g)|AO>

atm [int32 ndarray] libcint integral function argument

bas [int32 ndarray] libcint integral function argument

env [float64 ndarray] libcint integral function argument

coords [2D array, shape (N,3)] The coordinates of the grids.

Kwargs:

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.

non0tab [2D bool array] mask array to indicate whether the AO values are zero. The mask array can be obtained by calling `make_mask()`

out [ndarray] If provided, results are written into this array.

Returns: 2D array of shape (N,nao) Or 3D array of shape (*,N,nao) for AO values

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
>>> coords = numpy.random.random((100,3))  # 100 random points
>>> ao_value = mol.eval_gto("GTOval_sph", coords)
>>> print(ao_value.shape)
(100, 24)
>>> ao_value = mol.eval_gto("GTOval_ig_sph", coords, comp=3)
>>> print(ao_value.shape)
(3, 100, 24)
```

expand_etb(*l*, *n*, *alpha*, *beta*)

Generate the exponents of even tempered basis for `Mole.basis`. .. math:

```
e = e^{-\alpha * \beta^{i-1}} for i = 1 .. n
```

Args:

l [int] Angular momentum

n [int] Number of GTOs

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etb(1, 3, 1.5, 2)
[[1, [6.0, 1]], [1, [3.0, 1]], [1, [1.5, 1]]]
```

expand_etbs(*etbs*)

Generate even tempered basis. See also *expand_etb()*

Args: etbs = [(l, n, alpha, beta), (l, n, alpha, beta),...]

Returns: Formated `basis`

Examples:

```
>>> gto.expand_etbs([(0, 2, 1.5, 2.), (1, 2, 1, 2.)])
[[0, [6.0, 1]], [0, [3.0, 1]], [1, [1., 1]], [1, [2., 1]]]
```

format_atom(*atom*, *origin=0*, *axes=None*, *unit='Ang'*)

Convert the input `Mole.atom` to the internal data format. Including, changing the nuclear charge to atom symbol, converting the coordinates to AU, rotate and shift the molecule. If the `atom` is a string, it takes ";" and "n" for the mark to separate atoms; "," and arbitrary length of blank space to spearate the individual terms for an atom. Blank line will be ignored.

Args:

atoms [list or str] the same to `Mole.atom`

Kwargs:

origin [ndarray] new axis origin.

axes [ndarray] (new_x, new_y, new_z), each entry is a length-3 array

unit [str or number] If unit is one of strings (B, b, Bohr, bohr, AU, au), the coordinates of the input atoms are the atomic unit; If unit is one of strings (A, a, Angstrom, angstrom, Ang, ang), the coordinates are in the unit of angstrom; If a number is given, the number are considered as the Bohr value (in angstrom), which should be around 0.53

Returns: "atoms" in the internal format as `_atom`

Examples:

```
>>> gto.format_atom('9,0,0,0; h@1 0 0 1', origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0]], ['H@1', [-1.0, -1.0, 0.0]]]
>>> gto.format_atom(['9,0,0,0', (1, (0, 0, 1))], origin=(1,1,1))
[['F', [-1.0, -1.0, -1.0]], ['H', [-1, -1, 0]]]
```

format_basis(*basis_tab*)

Convert the input `Mole.basis` to the internal data format.

‘‘**{ atom: [(l, ((-exp, c_1, c_2, ..),**

(-exp, c_1, c_2, ..))),

(l, ((-exp, c_1, c_2, ..), (-exp, c_1, c_2, ..)))], ... }‘‘

Args:

basis_tab [dict] Similar to `Mole.basis`, it **cannot** be a str

Returns: Formated `basis`

Examples:

```
>>> gto.format_basis({'H':'sto-3g', 'H^2': '3-21g'})
{'H': [[0,
    [3.4252509099999999, 0.15432897000000001],
    [0.62391373000000006, 0.53532813999999995],
    [0.16885539999999999, 0.44463454000000002]]],
 'H^2': [[0,
    [5.4471780000000001, 0.15628500000000001],
    [0.82454700000000003, 0.90469100000000002]],
    [0, [0.18319199999999999, 1.0]]]}
```

format_ecp(*ecp_tab*)

Convert the input `ecp` (dict) to the internal data format:

```
{ atom: (nelec,  # core electrons
```

((l, # l=-1 for UL, l>=0 for Ul to indicate |l><l|

(((exp_1, c_1), # for r^0

(exp_2, c_2), ...),

((exp_1, c_1), # for r^1 (exp_2, c_2), ...),

((exp_1, c_1), # for r^2 ...))))),

...}

gto_norm(*l*, *expnt*)

Normalized factor for GTO radial part $g = r^l e^{-\alpha r^2}$

$$\frac{1}{\sqrt{\int g^2 r^2 dr}} = \sqrt{\frac{2^{2l+3}(l+1)!(2a)^{l+1.5}}{(2l+2)!\sqrt{\pi}}}$$

Ref: H. B. Schlegel and M. J. Frisch, Int. J. Quant. Chem., 54(1995), 83-87.

Args:

l (int): angular momentum

expnt : exponent α

Returns: normalization factor

Examples:

```
>>> print gto_norm(0, 1)
2.5264751109842591
```

has_ecp()
Whether pesudo potential is used in the system.

intor(*intor*, *comp=1*, *hermi=0*, *aosym='s1'*, *out=None*, *shls_slice=None*)
Integral generator.

Args:

intor [str] Name of the 1e or 2e AO integrals. Ref to `getints()` for the complete list of available 1-electron integral names

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp_sph has 3 components.

hermi [int] Symmetry of the integrals

0 : no symmetry assumed (default)
1 : hermitian
2 : anti-hermitian

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> mol.intor('int1e_ipnuc_sph', comp=3) # <nabla i | V_nuc | j>
[[[ 0.          0.        ]
  [ 0.          0.        ]]
 [[ 0.          0.        ]
  [ 0.          0.        ]]
 [[ 0.10289944  0.48176097]
  [-0.48176097 -0.10289944]]]
>>> mol.intor('int1e_nuc_spinor')
[[-1.69771092+0.j  0.00000000+0.j -0.67146312+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -1.69771092+0.j  0.00000000+0.j -0.67146312+0.j]
 [-0.67146312+0.j  0.00000000+0.j -1.69771092+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.67146312+0.j  0.00000000+0.j -1.69771092+0.j]]
```

intor_asymmetric(*intor*, *comp=1*)
One-electron integral generator. The integrals are assumed to be anti-hermitian

Args:

intor [str] Name of the 1-electron integral. Ref to `getints()` for the complete list of available 1-electron integral names

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp has 3 components.

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> mol.intor_asymmetric('int1e_nuc_spinor')
[[-1.69771092+0.j  0.00000000+0.j  0.67146312+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -1.69771092+0.j  0.00000000+0.j  0.67146312+0.j]
 [-0.67146312+0.j  0.00000000+0.j -1.69771092+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.67146312+0.j  0.00000000+0.j -1.69771092+0.j]]
```

intor_by_shell(*intor*, *shells*, *comp=1*)

For given 2, 3 or 4 shells, interface for libcint to get 1e, 2e, 2-center-2e or 3-center-2e integrals

Args:

intor_name [str] See also `getints()` for the supported intor_name

shls [list of int] The AO shell-ids of the integrals

atm [int32 ndarray] libcint integral function argument

bas [int32 ndarray] libcint integral function argument

env [float64 ndarray] libcint integral function argument

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp has 3 components.

Returns: ndarray of 2-dim to 5-dim, depending on the integral type (1e, 2e, 3c-2e, 2c2e) and the value of comp

Examples: The gradients of the spherical 2e integrals

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> gto.getints_by_shell('int2e_ip1_sph', (0,1,0,1), mol._atm, mol._bas, mol.
↪_env, comp=3)
[[[[[-0.        ]]]]
  [[[[-0.        ]]]]
  [[[[-0.08760462]]]]]
```

intor_symmetric(*intor*, *comp=1*)

One-electron integral generator. The integrals are assumed to be hermitian

Args:

intor [str] Name of the 1-electron integral. Ref to `getints()` for the complete list of available 1-electron integral names

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp_sph has 3 components.

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> mol.intor_symmetric('int1e_nuc_spinor')
[[-1.69771092+0.j  0.00000000+0.j -0.67146312+0.j  0.00000000+0.j]
```

```
[ 0.00000000+0.j -1.69771092+0.j  0.00000000+0.j -0.67146312+0.j]
[-0.67146312+0.j  0.00000000+0.j -1.69771092+0.j  0.00000000+0.j]
[ 0.00000000+0.j -0.67146312+0.j  0.00000000+0.j -1.69771092+0.j]]
```

kernel (*dump_input=True*, *parse_arg=True*, *verbose=None*, *output=None*, *max_memory=None*, *atom=None*, *basis=None*, *unit=None*, *nucmod=None*, *ecp=None*, *charge=None*, *spin=None*, *symmetry=None*, *symmetry_subgroup=None*, *cart=None*)

Setup moleclue and initialize some control parameters. Whenever you change the value of the attributes of *Mole*, you need call this function to refresh the internal data of Mole.

Kwargs:

dump_input [bool] whether to dump the contents of input file in the output file

parse_arg [bool] whether to read the sys.argv and overwrite the relevant parameters

verbose [int] Print level. If given, overwrite `Mole.verbose`

output [str or None] Output file. If given, overwrite `Mole.output`

max_memory [int, float] Allowd memory in MB. If given, overwrite `Mole.max_memory`

atom [list or str] To define molecluar structure.

basis [dict or str] To define basis set.

nucmod [dict or str] Nuclear model. If given, overwrite `Mole.nucmod`

charge [int] Charge of molecule. It affects the electron numbers If given, overwrite `Mole.charge`

spin [int] 2S, num. alpha electrons - num. beta electrons If given, overwrite `Mole.spin`

symmetry [bool or str] Whether to use symmetry. If given a string of point group name, the given point group symmetry will be used.

loads (*molstr*)

Deserialize a str containing a JSON document to a Mole object.

nao_2c (*mol*)

Total number of contracted spinor GTOs for the given *Mole* object

nao_2c_range (*mol*, *bas_id0*, *bas_id1*)

Lower and upper boundary of contracted spinor basis functions associated with the given shell range

Args:

mol : *Mole* object

bas_id0 [int] start shell id, 0-based

bas_id1 [int] stop shell id, 0-based

Returns: tupel of start basis function id and the stop function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.nao_2c_range(mol, 2, 4)
(4, 12)
```

nao_cart (*mol*)

Total number of contracted cartesian GTOs for the given *Mole* object

nao_nr (*mol*, *cart=False*)

Total number of contracted spherical GTOs for the given *Mole* object

nao_nr_range(*mol*, *bas_id0*, *bas_id1*)
Lower and upper boundary of contracted spherical basis functions associated with the given shell range

Args:

mol : *Mole* object

bas_id0 [int] start shell id

bas_id1 [int] stop shell id

Returns: tupel of start basis function id and the stop function id

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; C 0 0 1', basis='6-31g')
>>> gto.nao_nr_range(mol, 2, 4)
(2, 6)
```

npgto_nr(*mol*, *cart=False*)
Total number of primitive spherical GTOs for the given *Mole* object

offset_2c_by_atom(*mol*)
2-component AO offset for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

offset_ao_by_atom(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

offset_nr_by_atom(*mol*, *ao_loc=None*)
AO offsets for each atom. Return a list, each item of the list gives (start-shell-id, stop-shell-id, start-AO-id, stop-AO-id)

pack(*mol*)
Pack the input args of *Mole* to a dict.

Note this function only pack the input arguments (not the entire Mole class). Modifications to mol._atm, mol._bas, mol._env are not tracked. Use *dumps()* to serialize the entire Mole object.

search_ao_label(*mol*, *label*)
Find the index of the AO basis function based on the given ao_label

Args:

ao_label [string or a list of strings] The regular expression pattern to match the orbital labels returned by mol.ao_labels()

Returns: A list of index for the AOs that matches the given ao_label RE pattern

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='ccpvtz')
>>> mol.parse_aolabel('Cl.*p')
[19 20 21 22 23 24 25 26 27 28 29 30]
>>> mol.parse_aolabel('Cl 2p')
[19 20 21]
>>> mol.parse_aolabel(['Cl.*d', 'Cl 4p'])
[25 26 27 31 32 33 34 35 36 37 38 39 40]
```

search_ao_nr(*mol*, *atm_id*, *l*, *m*, *atmshell*)
Search the first basis function id (**not** the shell id) which matches the given atom-id, angular momentum magnetic angular momentum, principal shell.

Args:

atm_id [int] atom id, 0-based

l [int] angular momentum

m [int] magnetic angular momentum

atmshell [int] principal quantum number

Returns: basis function id, 0-based. If not found, return None

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> mol.search_ao_nr(1, 1, -1, 3) # Cl 3px
7
```

set_common_orig(*coord*)

Update common origin which held in :class‘Mole‘._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_common_orig_(*coord*)

Update common origin which held in :class‘Mole‘._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_common_origin(*coord*)

Update common origin which held in :class‘Mole‘._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_common_origin_(*coord*)

Update common origin which held in :class‘Mole‘._env. **Note** the unit is Bohr

Examples:

```
>>> mol.set_common_orig(0)
>>> mol.set_common_orig((1,0,0))
```

set_f12_zeta(*zeta*)

Set zeta for YP exp(-zeta r12)/r12 or STG exp(-zeta r12) type integrals

set_geom_(*atoms*, *unit='Angstrom'*, *symmetry=None*)

Replace geometry

set_nuc_mod(*atm_id*, *zeta*)

Change the nuclear charge distribution of the given atom ID. The charge distribution is defined as: rho(r) = nuc_charge * Norm * exp(-zeta * r^2). This function can **only** be called after .build() method is executed.

Examples:

```
>>> for ia in range(mol.natm):
...     zeta = gto.filatov_nuc_mod(mol.atom_charge(ia))
...     mol.set_nuc_mod(ia, zeta)
```

set_nuc_mod_(*atm_id*, *zeta*)
Change the nuclear charge distribution of the given atom ID. The charge distribution is defined as: rho(r) = nuc_charge * Norm * exp(-zeta * r^2). This function can **only** be called after .build() method is executed.

Examples:

```
>>> for ia in range(mol.natm):
...     zeta = gto.filatov_nuc_mod(mol.atom_charge(ia))
...     mol.set_nuc_mod(ia, zeta)
```

set_range_coulomb(*omega*)
Apply the long range part of range-separated Coulomb operator for **all** 2e integrals erf(omega r12) / r12 set omega to 0 to siwtch off the range-separated Coulomb

set_range_coulomb_(*omega*)
Apply the long range part of range-separated Coulomb operator for **all** 2e integrals erf(omega r12) / r12 set omega to 0 to siwtch off the range-separated Coulomb

set_rinv_orig(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_orig_(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_origin(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_origin_(*coord*)
Update origin for operator $\frac{1}{|r-R_O|}$. **Note** the unit is Bohr

Examples:

```
>>> mol.set_rinv_orig(0)
>>> mol.set_rinv_orig((0,1,0))
```

set_rinv_zeta(*zeta*)
Assume the charge distribution on the "rinv_orig". zeta is the parameter to control the charge distribution: rho(r) = Norm * exp(-zeta * r^2). **Be careful** when call this function. It affects the behavior of int1e_rinv_* functions. Make sure to set it back to 0 after using it!

set_rinv_zeta_(*zeta*)
Assume the charge distribution on the "rinv_orig". zeta is the parameter to control the charge distribution: rho(r) = Norm * exp(-zeta * r^2). **Be careful** when call this function. It affects the behavior of int1e_rinv_* functions. Make sure to set it back to 0 after using it!

spheric_labels(*fmt=False*)
Labels for spherical GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-real-spherical-notation] or formatted strings based on the argument "fmt"

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.spherical_labels(mol)
[(0, 'H', '1s', ''), (1, 'Cl', '1s', ''), (1, 'Cl', '2s', ''), (1, 'Cl', '3s
↪', ''), (1, 'Cl', '2p', 'x'), (1, 'Cl', '2p', 'y'), (1, 'Cl', '2p', 'z'),␣
↪(1, 'Cl', '3p', 'x'), (1, 'Cl', '3p', 'y'), (1, 'Cl', '3p', 'z')]
```

spherical_labels(*fmt=False*)
Labels for spherical GTO functions

Kwargs: fmt : str or bool if fmt is boolean, it controls whether to format the labels and the default format is "%d%3s %s%-4s". if fmt is string, the string will be used as the print format.

Returns: List of [(atom-id, symbol-str, nl-str, str-of-real-spherical-notation] or formatted strings based on the argument "fmt"

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; Cl 0 0 1', basis='sto-3g')
>>> gto.spherical_labels(mol)
[(0, 'H', '1s', ''), (1, 'Cl', '1s', ''), (1, 'Cl', '2s', ''), (1, 'Cl', '3s
↪', ''), (1, 'Cl', '2p', 'x'), (1, 'Cl', '2p', 'y'), (1, 'Cl', '2p', 'z'),␣
↪(1, 'Cl', '3p', 'x'), (1, 'Cl', '3p', 'y'), (1, 'Cl', '3p', 'z')]
```

time_reversal_map(*mol*)
The index to map the spinor functions and its time reversal counterpart. The returned indices have postive or negative values. For the i-th basis function, if the returned j = idx[i] < 0, it means $T|i\rangle = -|j\rangle$, otherwise $T|i\rangle = |j\rangle$

tot_electrons(*mol*)
Total number of electrons for the given molecule

Returns: electron number in integer

Examples:

```
>>> mol = gto.M(atom='H 0 1 0; C 0 0 1', charge=1)
>>> mol.tot_electrons()
6
```

unpack(*moldic*)
Unpack a dict which is packed by *pack()*, to generate the input arguments for *Mole* object.

moleintor

pyscf.gto.moleintor.**ascint3**(*intor_name*)
convert cint2 function name to cint3 function name

pyscf.gto.moleintor.**getints**(*intor_name*, *atm*, *bas*, *env*, *shls_slice=None*, *comp=1*, *hermi=0*, *aosym='s1'*, *ao_loc=None*, *cintopt=None*, *out=None*)
1e and 2e integral generator.

Args: intor_name : str

Function	Expression
"int1e_ovlp_sph"	(|)
"int1e_nuc_sph"	(| nuc |)
"int1e_kin_sph"	(.5 | p dot p)
"int1e_ia01p_sph"	(#C(0 1) | nabla-rinv | cross p)
"int1e_giao_irjxp_sph"	(#C(0 1) | r cross p)
"int1e_cg_irxp_sph"	(#C(0 1) | rc cross p)
"int1e_giao_a11part_sph"	(-.5 | nabla-rinv | r)
"int1e_cg_a11part_sph"	(-.5 | nabla-rinv | rc)
"int1e_a01gp_sph"	(g | nabla-rinv cross p |)
"int1e_igkin_sph"	(#C(0 .5) g | p dot p)
"int1e_igovlp_sph"	(#C(0 1) g |)
"int1e_ignuc_sph"	(#C(0 1) g | nuc |)
"int1e_z_sph"	(| zc |)
"int1e_zz_sph"	(| zc zc |)
"int1e_r_sph"	(| rc |)
"int1e_r2_sph"	(| rc dot rc |)
"int1e_rr_sph"	(| rc rc |)
"int1e_pnucp_sph"	(p* | nuc dot p |)
"int1e_prinvxp_sph"	(p* | rinv cross p |)
"int1e_ovlp_spinor"	(|)
"int1e_nuc_spinor"	(| nuc |)
"int1e_srsr_spinor"	(sigma dot r | sigma dot r)
"int1e_sr_spinor"	(sigma dot r |)
"int1e_srsp_spinor"	(sigma dot r | sigma dot p)
"int1e_spsp_spinor"	(sigma dot p | sigma dot p)
"int1e_sp_spinor"	(sigma dot p |)
"int1e_spnucsp_spinor"	(sigma dot p | nuc | sigma dot p)
"int1e_srnucsr_spinor"	(sigma dot r | nuc | sigma dot r)
"int1e_govlp_spinor"	(g |)
"int1e_gnuc_spinor"	(g | nuc |)
"int1e_cg_sa10sa01_spinor"	(.5 sigma cross rc | sigma cross nabla-rinv |)
"int1e_cg_sa10sp_spinor"	(.5 rc cross sigma | sigma dot p)
"int1e_cg_sa10nucsp_spinor"	(.5 rc cross sigma | nuc | sigma dot p)
"int1e_giao_sa10sa01_spinor"	(.5 sigma cross r | sigma cross nabla-rinv |)
"int1e_giao_sa10sp_spinor"	(.5 r cross sigma | sigma dot p)
"int1e_giao_sa10nucsp_spinor"	(.5 r cross sigma | nuc | sigma dot p)
"int1e_sa01sp_spinor"	(| nabla-rinv cross sigma | sigma dot p)
"int1e_spgsp_spinor"	(g sigma dot p | sigma dot p)
"int1e_spgnucsp_spinor"	(g sigma dot p | nuc | sigma dot p)
"int1e_spgsa01_spinor"	(g sigma dot p | nabla-rinv cross sigma |)
"int1e_spspsp_spinor"	(sigma dot p | sigma dot p sigma dot p)

Continued on next page

Table 1.1 – continued from previous page

Function	Expression
“int1e_spnuc_spinor”	(sigma dot p | nuc |)
“int1e_ovlp_cart”	(|)
“int1e_nuc_cart”	(| nuc |)
“int1e_kin_cart”	(.5 | p dot p)
“int1e_ia01p_cart”	(#C(0 1) | nabla-rinv | cross p)
“int1e_giao_irjxp_cart”	(#C(0 1) | r cross p)
“int1e_cg_irxp_cart”	(#C(0 1) | rc cross p)
“int1e_giao_a11part_cart”	(-.5 | nabla-rinv | r)
“int1e_cg_a11part_cart”	(-.5 | nabla-rinv | rc)
“int1e_a01gp_cart”	(g | nabla-rinv cross p |)
“int1e_igkin_cart”	(#C(0 .5) g | p dot p)
“int1e_igovlp_cart”	(#C(0 1) g |)
“int1e_ignuc_cart”	(#C(0 1) g | nuc |)
“int1e_ipovlp_sph”	(nabla |)
“int1e_ipkin_sph”	(.5 nabla | p dot p)
“int1e_ipnuc_sph”	(nabla | nuc |)
“int1e_iprinv_sph”	(nabla | rinv |)
“int1e_rinv_sph”	(| rinv |)
“int1e_ipovlp_spinor”	(nabla |)
“int1e_ipkin_spinor”	(.5 nabla | p dot p)
“int1e_ipnuc_spinor”	(nabla | nuc |)
“int1e_iprinv_spinor”	(nabla | rinv |)
“int1e_ipspnucsp_spinor”	(nabla sigma dot p | nuc | sigma dot p)
“int1e_ipsprinvsp_spinor”	(nabla sigma dot p | rinv | sigma dot p)
“int1e_ipovlp_cart”	(nabla |)
“int1e_ipkin_cart”	(.5 nabla | p dot p)
“int1e_ipnuc_cart”	(nabla | nuc |)
“int1e_iprinv_cart”	(nabla | rinv |)
“int1e_rinv_cart”	(| rinv |)
“int2e_p1vxp1_sph”	(p* , cross p | ,) ; SSO
“int2e_sph”	(, | ,)
“int2e_ig1_sph”	(#C(0 1) g , | ,)
“int2e_spinor”	(, | ,)
“int2e_spsp1_spinor”	(sigma dot p , sigma dot p | ,)
“int2e_spsp1spsp2_spinor”	(sigma dot p , sigma dot p | sigma dot p , sigma dot p)
“int2e_srsr1_spinor”	(sigma dot r , sigma dot r | ,)
“int2e_srsr1srsr2_spinor”	(sigma dot r , sigma dot r | sigma dot r , sigma dot r)
“int2e_cg_sa10sp1_spinor”	(.5 rc cross sigma , sigma dot p | ,)
“int2e_cg_sa10sp1spsp2_spinor”	(.5 rc cross sigma , sigma dot p | sigma dot p , sigma dot p)
“int2e_giao_sa10sp1_spinor”	(.5 r cross sigma , sigma dot p | ,)
“int2e_giao_sa10sp1spsp2_spinor”	(.5 r cross sigma , sigma dot p | sigma dot p , sigma dot p)
“int2e_g1_spinor”	(g , | ,)
“int2e_spgsp1_spinor”	(g sigma dot p , sigma dot p | ,)
“int2e_g1spsp2_spinor”	(g , | sigma dot p , sigma dot p)
“int2e_spgsp1spsp2_spinor”	(g sigma dot p , sigma dot p | sigma dot p , sigma dot p)
“int2e_spv1_spinor”	(sigma dot p , | ,)
“int2e_vsp1_spinor”	(, sigma dot p | ,)
“int2e_spsp2_spinor”	(, | sigma dot p , sigma dot p)
“int2e_spv1spv2_spinor”	(sigma dot p , | sigma dot p ,)

Continued on next page

Table 1.1 – continued from previous page

Function	Expression
“int2e_vsp1spv2_spinor”	(, sigma dot p | sigma dot p ,)
“int2e_spv1vsp2_spinor”	(sigma dot p , | , sigma dot p)
“int2e_vsp1vsp2_spinor”	(, sigma dot p | , sigma dot p)
“int2e_spv1spsp2_spinor”	(sigma dot p , | sigma dot p , sigma dot p)
“int2e_vsp1spsp2_spinor”	(, sigma dot p | sigma dot p , sigma dot p)
“int2e_ig1_cart”	(#C(0 1) g , | ,)
“int2e_ip1_sph”	(nabla , | ,)
“int2e_ip1_spinor”	(nabla , | ,)
“int2e_ipspsp1_spinor”	(nabla sigma dot p , sigma dot p | ,)
“int2e_ip1spsp2_spinor”	(nabla , | sigma dot p , sigma dot p)
“int2e_ipspsp1spsp2_spinor”	(nabla sigma dot p , sigma dot p | sigma dot p , sigma dot p)
“int2e_ipsrsr1_spinor”	(nabla sigma dot r , sigma dot r | ,)
“int2e_ip1srsr2_spinor”	(nabla , | sigma dot r , sigma dot r)
“int2e_ipsrsr1srsr2_spinor”	(nabla sigma dot r , sigma dot r | sigma dot r , sigma dot r)
“int2e_ip1_cart”	(nabla , | ,)
“int2e_ssp1ssp2_spinor”	(, sigma dot p | gaunt | , sigma dot p)
“int2e_cg_ssa10ssp2_spinor”	(rc cross sigma , | gaunt | , sigma dot p)
“int2e_giao_ssa10ssp2_spinor”	(r cross sigma , | gaunt | , sigma dot p)
“int2e_gssp1ssp2_spinor”	(g , sigma dot p | gaunt | , sigma dot p)
“int2e_ipip1_sph”	(nabla nabla , | ,)
“int2e_ipvip1_sph”	(nabla , nabla | ,)
“int2e_ip1ip2_sph”	(nabla , | nabla ,)
“int3c2e_ip1_sph”	(nabla , |)
“int3c2e_ip2_sph”	(, | nabla)
“int2c2e_ip1_sph”	(nabla | r12 |)
“int3c2e_spinor”	(nabla , |)
“int3c2e_spsp1_spinor”	(nabla , |)
“int3c2e_ip1_spinor”	(nabla , |)
“int3c2e_ip2_spinor”	(, | nabla)
“int3c2e_ipspsp1_spinor”	(nabla sigma dot p , sigma dot p |)
“int3c2e_spsp1ip2_spinor”	(sigma dot p , sigma dot p | nabla)

atm [int32 ndarray] libcint integral function argument

bas [int32 ndarray] libcint integral function argument

env [float64 ndarray] libcint integral function argument

Kwargs:

shls_slice [8-element list] (ish_start, ish_end, jsh_start, jsh_end, ksh_start, ksh_end, lsh_start, lsh_end)

comp [int] Components of the integrals, e.g. int1e_ipovlp has 3 components.

hermi [int (1e integral only)] Symmetry of the 1e integrals

0 : no symmetry assumed (default)
1 : hermitian
2 : anti-hermitian

aosym [str (2e integral only)] Symmetry of the 2e integrals

4 or '4' or 's4': 4-fold symmetry (default)
'2ij' or 's2ij' : symmetry between i, j in (ij|kl)
'2kl' or 's2kl' : symmetry between k, l in (ij|kl)
1 or '1' or 's1': no symmetry

out [ndarray (2e integral only)] array to store the 2e AO integrals

Returns: ndarray of 1-electron integrals, can be either 2-dim or 3-dim, depending on comp

Examples:

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> gto.getints('int1e_ipnuc_sph', mol._atm, mol._bas, mol._env, comp=3) # <nabla␣
↪i | V_nuc | j>
[[[ 0.          0.        ]
  [ 0.          0.        ]]
 [[ 0.          0.        ]
  [ 0.          0.        ]]
 [[ 0.10289944  0.48176097]
  [-0.48176097 -0.10289944]]]
```

pyscf.gto.moleintor.**getints_by_shell**(*intor_name*, *shls*, *atm*, *bas*, *env*, *comp=1*)

For given 2, 3 or 4 shells, interface for libcint to get 1e, 2e, 2-center-2e or 3-center-2e integrals

Args:

intor_name [str] See also *getints()* for the supported intor_name

shls [list of int] The AO shell-ids of the integrals

atm [int32 ndarray] libcint integral function argument

bas [int32 ndarray] libcint integral function argument

env [float64 ndarray] libcint integral function argument

Kwargs:

comp [int] Components of the integrals, e.g. int1e_ipovlp has 3 components.

Returns: ndarray of 2-dim to 5-dim, depending on the integral type (1e, 2e, 3c-2e, 2c2e) and the value of comp

Examples: The gradients of the spherical 2e integrals

```
>>> mol.build(atom='H 0 0 0; H 0 0 1.1', basis='sto-3g')
>>> gto.getints_by_shell('int2e_ip1_sph', (0,1,0,1), mol._atm, mol._bas, mol._
↪env, comp=3)
[[[[[-0.        ]]]]
  [[[[-0.        ]]]]
  [[[[-0.08760462]]]]]
```

basis

Internal format

This module loads basis set and ECP data from basis database and parse the basis (mostly in NWChem format) and finally convert to internal format. The internal format of basis set is:

```
basis = {atom_type1:[[angular_momentum
                      (GTO-exp1, contract-coeff11, contract-coeff12),
                      (GTO-exp2, contract-coeff21, contract-coeff22),
                      (GTO-exp3, contract-coeff31, contract-coeff32),
                      ...],
                     [angular_momentum
                      (GTO-exp1, contract-coeff11, contract-coeff12),
                      ...],
                     ...],
         atom_type2:[[angular_momentum, (...),],
                      ...],
```

For example:

```
mol.basis = {'H': [[0,
                    (19.2406000, 0.0328280),
                    (2.8992000, 0.2312080),
                    (0.6534000, 0.8172380),],
                   [0,
                    (0.1776000, 1.0000000),],
                   [1,
                    (1.0000000, 1.0000000),]],
             }
```

Some basis sets, e.g. `pyscf/gto/basis/dzp_dunning.py`, are saved in the internal format.

`pyscf.gto.basis.`**`load`**(*filename_or_basisname*, *symb*)

Convert the basis of the given symbol to internal format

Args:

filename_or_basisname [str] Case insensitive basis set name. Special characters will be removed. or a string of "path/to/file" which stores the basis functions

symb [str] Atomic symbol, Special characters will be removed.

Examples: Load STO 3G basis of carbon to oxygen atom

```
>>> mol = gto.Mole()
>>> mol.basis = {'O': load('sto-3g', 'C')}
```

`pyscf.gto.basis.`**`load_ecp`**(*filename_or_basisname*, *symb*)

Convert the basis of the given symbol to internal format

`pyscf.gto.basis.`**`parse`**(*string*, *symb=None*)

Parse the NWChem format basis or ECP text, return an internal basis (ECP) format which can be assigned to `Mole.basis` or `Mole.ecp`

Args: string : Blank linke and the lines of "BASIS SET" and "END" will be ignored

Examples:

```
>>> mol = gto.Mole()
>>> mol.basis = {'O': gto.basis.parse("""
... #BASIS SET: (6s,3p) -> [2s,1p]
... 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
... """)}
```

1.5 lib — Helper functions, parameters, and C extensions

C code and some fundamental functions

1.5.1 parameters

Some PySCF environment parameters are defined in this module.

Scratch directory

The PySCF scratch directory is specified by `TMPDIR`. Its default value is the same to the system environment variable `TMPDIR`. It can be overwritten by the system environment variable `PYSCF_TMPDIR`.

Maximum memory

The variable `MAX_MEMORY` defines the maximum memory that PySCF can be used in the calculation. Its unit is MB. The default value is 4000 MB. It can be overwritten by the system environment variable `PYSCF_MAX_MEMORY`.

Note: Some calculations may exceed the max_memory limit, especially when the `incore_anyway` of `Mole` object was set.

1.5.2 logger

Logging system

Log level

Level	number
DEBUG4	9
DEBUG3	8
DEBUG2	7
DEBUG1	6
DEBUG	5
INFO	4
NOTE	3
WARN	2
ERROR	1
QUIET	0

Big `verbose` value means more noise in the output file.

Note: At log level 1 (ERROR) and 2 (WARN), the messages are also output to stderr.

Each Logger object has its own output destination and verbose level. So multiple Logger objects can be created to manage the message system without affecting each other. The methods provided by Logger class has the direct connection to the log level. E.g. `info()` print messages if the verbose level >= 4 (INFO):

```
>>> import sys
>>> from pyscf import lib
>>> log = lib.logger.Logger(sys.stdout, 4)
>>> log.info('info level')
info level
>>> log.verbose = 3
>>> log.info('info level')
>>> log.note('note level')
note level
```

timer

Logger object provides timer method for timing. Set `TIMER_LEVEL` to control which level to output the timing. It is 5 (DEBUG) by default.

```
>>> import sys, time
>>> from pyscf import lib
>>> log = lib.logger.Logger(sys.stdout, 4)
>>> t0 = time.clock()
>>> log.timer('test', t0)
>>> lib.logger.TIMER_LEVEL = 4
>>> log.timer('test', t0)
    CPU time for test      0.00 sec
```

1.5.3 numpy helper

pyscf.lib.numpy_helper.**asarray**(*a*, *dtype=None*, *order=None*)
Convert a list of N-dim arrays to a (N+1) dim array. It is equivalent to numpy.asarray function but more efficient.

pyscf.lib.numpy_helper.**cartesian_prod**(*arrays*, *out=None*)
Generate a cartesian product of input arrays. http://stackoverflow.com/questions/1208118/using-numpy-to-build-an-array-of-all-combinations-of-two-arrays

Args:

arrays [list of array-like] 1-D arrays to form the cartesian product of.

out [ndarray] Array to place the cartesian product in.

Returns:

out [ndarray] 2-D array of shape (M, len(arrays)) containing cartesian products formed of input arrays.

Examples:

```
>>> cartesian_prod(([1, 2, 3], [4, 5], [6, 7]))
array([[1, 4, 6],
       [1, 4, 7],
       [1, 5, 6],
       [1, 5, 7],
       [2, 4, 6],
       [2, 4, 7],
       [2, 5, 6],
       [2, 5, 7],
       [3, 4, 6],
       [3, 4, 7],
       [3, 5, 6],
       [3, 5, 7]])
```

pyscf.lib.numpy_helper.**cond**(*x*, *p=None*)
Compute the condition number

pyscf.lib.numpy_helper.**condense**(*opname*, *a*, *locs*)

```
nd = loc[-1]
out = numpy.empty((nd,nd))
for i,i0 in enumerate(loc):
    i1 = loc[i+1]
    for j,j0 in enumerate(loc):
        j1 = loc[j+1]
        out[i,j] = op(a[i0:i1,j0:j1])
return out
```

pyscf.lib.numpy_helper.**ddot**(*a*, *b*, *alpha=1*, *c=None*, *beta=0*)
Matrix-matrix multiplication for double precision arrays

pyscf.lib.numpy_helper.**direct_sum**(*subscripts*, **operands*)
Apply the summation over many operands with the einsum fashion.

Examples:

```
>>> a = numpy.random((6,5))
>>> b = numpy.random((4,3,2))
>>> direct_sum('ij,klm->ijklm', a, b).shape
```

```
(6, 5, 4, 3, 2)
>>> direct_sum('ij,klm', a, b).shape
(6, 5, 4, 3, 2)
>>> direct_sum('i,j,klm->mjlik', a[0], a[:,0], b).shape
(2, 6, 3, 5, 4)
>>> direct_sum('ij-klm->ijklm', a, b).shape
(6, 5, 4, 3, 2)
>>> direct_sum('ij+klm', a, b).shape
(6, 5, 4, 3, 2)
>>> direct_sum('-i-j+klm->mjlik', a[0], a[:,0], b).shape
(2, 6, 3, 5, 4)
>>> c = numpy.random((3,5))
>>> z = direct_sum('ik+jk->kij', a, c).shape  # This is slow
>>> abs(a.T.reshape(5,6,1) + c.reshape(5,1,3) - z).sum()
0.0
```

pyscf.lib.numpy_helper.**einsum**(*idx_str*, **tensors*)
Perform a more efficient einsum via reshaping to a matrix multiply.

Current differences compared to numpy.einsum: This assumes that each repeated index is actually summed (i.e. no 'i,i->i') and appears only twice (i.e. no 'ij,ik,il->jkl'). The output indices must be explicitly specified (i.e. 'ij,j->i' and not 'ij,j').

pyscf.lib.numpy_helper.**hermi_sum**(*a*, *axes=None*, *hermi=1*, *inplace=False*, *out=None*)
a + a.T for better memory efficiency

Examples:

```
>>> transpose_sum(numpy.arange(4.).reshape(2,2))
[[ 0.  3.]
 [ 3.  6.]]
```

pyscf.lib.numpy_helper.**hermi_triu**(*mat*, *hermi=1*, *inplace=True*)
Use the elements of the lower triangular part to fill the upper triangular part.

Kwargs: filltriu : int

1 (default) return a hermitian matrix

2 return an anti-hermitian matrix

Examples:

```
>>> unpack_row(numpy.arange(9.).reshape(3,3), 1)
[[ 0.  3.  6.]
 [ 3.  4.  7.]
 [ 6.  7.  8.]]
>>> unpack_row(numpy.arange(9.).reshape(3,3), 2)
[[ 0. -3. -6.]
 [ 3.  4. -7.]
 [ 6.  7.  8.]]
```

pyscf.lib.numpy_helper.**norm**(*x*, *ord=None*, *axis=None*)
numpy.linalg.norm for numpy 1.6.*

pyscf.lib.numpy_helper.**pack_tril**(*mat*, *axis=-1*, *out=None*)
flatten the lower triangular part of a matrix. Given mat, it returns mat[...,numpy.tril_indices(mat.shape[0])]

Examples:

```
>>> pack_tril(numpy.arange(9).reshape(3,3))
[0 3 4 6 7 8]
```

pyscf.lib.numpy_helper.**solve_lineq_by_SVD**(*a*, *b*)
a * x = b

pyscf.lib.numpy_helper.**take_2d**(*a*, *idx*, *idy*, *out=None*)
a(idx,idy)

Examples:

```
>>> out = numpy.arange(9.).reshape(3,3)
>>> take_2d(a, [0,2], [0,2])
[[ 0.  2.]
 [ 6.  8.]]
```

pyscf.lib.numpy_helper.**takebak_2d**(*out*, *a*, *idx*, *idy*)
Reverse operation of take_2d. out(idx,idy) += a

Examples:

```
>>> out = numpy.zeros((3,3))
>>> takebak_2d(out, numpy.ones((2,2)), [0,2], [0,2])
[[ 1.  0.  1.]
 [ 0.  0.  0.]
 [ 1.  0.  1.]]
```

pyscf.lib.numpy_helper.**transpose**(*a*, *axes=None*, *inplace=False*, *out=None*)
Transpose array for better memory efficiency

Examples:

```
>>> transpose(numpy.ones((3,2)))
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
```

pyscf.lib.numpy_helper.**transpose_sum**(*a*, *inplace=False*, *out=None*)
a + a.T for better memory efficiency

Examples:

```
>>> transpose_sum(numpy.arange(4.).reshape(2,2))
[[ 0.  3.]
 [ 3.  6.]]
```

pyscf.lib.numpy_helper.**unpack_row**(*tril*, *row_id*)
Extract one row of the lower triangular part of a matrix. It is equivalent to unpack_tril(a)[row_id]

Examples:

```
>>> unpack_row(numpy.arange(6.), 0)
[ 0. 1. 3.]
>>> unpack_tril(numpy.arange(6.))[0]
[ 0. 1. 3.]
```

pyscf.lib.numpy_helper.**unpack_tril**(*tril*, *filltriu=1*, *axis=-1*, *out=None*)
Reverse operation of pack_tril.

Kwargs: filltriu : int

0 Do not fill the upper triangular part, random number may appear in the upper triangular part

1 (default) Transpose the lower triangular part to fill the upper triangular part

2 Similar to filltriu=1, negative of the lower triangular part is assign to the upper triangular part to make the matrix anti-hermitian

Examples:

```
>>> unpack_tril(numpy.arange(6.))
[[ 0. 1. 3.]
 [ 1. 2. 4.]
 [ 3. 4. 5.]]
>>> unpack_tril(numpy.arange(6.), 0)
[[ 0. 0. 0.]
 [ 1. 2. 0.]
 [ 3. 4. 5.]]
>>> unpack_tril(numpy.arange(6.), 2)
[[ 0. -1. -3.]
 [ 1.  2. -4.]
 [ 3.  4.  5.]]
```

pyscf.lib.numpy_helper.**zdot**(*a*, *b*, *alpha=1*, *c=None*, *beta=0*)

Matrix-matrix multiplication for double complex arrays using Gauss's complex multiplication algorithm

Extension to scipy.linalg module

pyscf.lib.linalg_helper.**cho_solve**(*a*, *b*)

Solve ax = b, where a is hermitian matrix

pyscf.lib.linalg_helper.**davidson**(*aop*, *x0*, *precond*, *tol=1e-12*, *max_cycle=50*, *max_space=12*, *lindep=1e-14*, *max_memory=2000*, *dot=<built-in function dot>*, *callback=None*, *nroots=1*, *lessio=False*, *verbose=2*, *follow_state=False*)

Davidson diagonalization method to solve a c = e c. Ref [1] E.R. Davidson, J. Comput. Phys. 17 (1), 87-94 (1975). [2] http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter11.pdf

Args:

aop [function(x) => array_like_x] aop(x) to mimic the matrix vector multiplication $\sum_j a_{ij} * x_j$. The argument is a 1D array. The returned value is a 1D array.

x0 [1D array or a list of 1D array] Initial guess. The initial guess vector(s) are just used as the initial subspace bases. If the subspace is smaller than "nroots", eg 10 roots and one initial guess, all eigenvectors are chosen as the eigenvectors during the iterations. The first iteration has one eigenvector, the next iteration has two, the third iterastion has 4, ..., until the subspace size > nroots.

precond [function(dx, e, x0) => array_like_dx] Preconditioner to generate new trial vector. The argument dx is a residual vector `a*x0-e*x0`; e is the current eigenvalue; x0 is the current eigenvector.

Kwargs:

tol [float] Convergence tolerance.

max_cycle [int] max number of iterations.

max_space [int] space size to hold trial vectors.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

max_memory [int or float] Allowed memory in MB.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

nroots [int] Number of eigenvalues to be computed. When nroots > 1, it affects the shape of the return value

lessio [bool] How to compute a*x0 for current eigenvector x0. There are two ways to compute a*x0. One is to assemble the existed a*x. The other is to call aop(x0). The default is the first method which needs more IO and less computational cost. When IO is slow, the second method can be considered.

follow_state [bool] If the solution dramatically changes in two iterations, clean the subspace and restart the iteration with the old solution. It can help to improve numerical stability. Default is False.

Returns:

e [float or list of floats] Eigenvalue. By default it's one float number. If `nroots` > 1, it is a list of floats for the lowest `nroots` eigenvalues.

c [1D array or list of 1D arrays] Eigenvector. By default it's a 1D array. If `nroots` > 1, it is a list of arrays for the lowest `nroots` eigenvectors.

Examples:

```
>>> from pyscf import lib
>>> a = numpy.random.random((10,10))
>>> a = a + a.T
>>> aop = lambda x: numpy.dot(a,x)
>>> precond = lambda dx, e, x0: dx/(a.diagonal()-e)
>>> x0 = a[0]
>>> e, c = lib.davidson(aop, x0, precond)
```

pyscf.lib.linalg_helper.**davidson1**(*aop*, *x0*, *precond*, *tol=1e-12*, *max_cycle=50*, *max_space=12*, *lindep=1e-14*, *max_memory=2000*, *dot=<built-in function dot>*, *callback=None*, *nroots=1*, *lessio=False*, *verbose=2*, *follow_state=False*)

Davidson diagonalization method to solve a c = e c. Ref [1] E.R. Davidson, J. Comput. Phys. 17 (1), 87-94 (1975). [2] http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter11.pdf

Args:

aop [function([x]) => [array_like_x]] Matrix vector multiplication $y_{ki} = \sum_j a_{ij} * x_{jk}$.

x0 [1D array or a list of 1D array] Initial guess. The initial guess vector(s) are just used as the initial subspace bases. If the subspace is smaller than "nroots", eg 10 roots and one initial guess, all eigenvectors are chosen as the eigenvectors during the iterations. The first iteration has one eigenvector, the next iteration has two, the third iterastion has 4, ..., until the subspace size > nroots.

precond [function(dx, e, x0) => array_like_dx] Preconditioner to generate new trial vector. The argument dx is a residual vector `a*x0-e*x0`; e is the current eigenvalue; x0 is the current eigenvector.

Kwargs:

tol [float] Convergence tolerance.

max_cycle [int] max number of iterations.

max_space [int] space size to hold trial vectors.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

max_memory [int or float] Allowed memory in MB.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

nroots [int] Number of eigenvalues to be computed. When nroots > 1, it affects the shape of the return value

lessio [bool] How to compute a*x0 for current eigenvector x0. There are two ways to compute a*x0. One is to assemble the existed a*x. The other is to call aop(x0). The default is the first method which needs more IO and less computational cost. When IO is slow, the second method can be considered.

follow_state [bool] If the solution dramatically changes in two iterations, clean the subspace and restart the iteration with the old solution. It can help to improve numerical stability. Default is False.

Returns:

conv [bool] Converged or not

e [list of floats] The lowest `nroots` eigenvalues.

c [list of 1D arrays] The lowest `nroots` eigenvectors.

Examples:

```
>>> from pyscf import lib
>>> a = numpy.random.random((10,10))
>>> a = a + a.T
>>> aop = lambda xs: [numpy.dot(a,x) for x in xs]
>>> precond = lambda dx, e, x0: dx/(a.diagonal()-e)
>>> x0 = a[0]
>>> e, c = lib.davidson(aop, x0, precond, nroots=2)
>>> len(e)
2
```

pyscf.lib.linalg_helper.**davidson_nosym**(*aop*, *x0*, *precond*, *tol=1e-12*, *max_cycle=50*, *max_space=12*, *lindep=1e-14*, *max_memory=2000*, *dot=<built-in function dot>*, *callback=None*, *nroots=1*, *lessio=False*, *left=False*, *pick=<function pick_real_eigs>*, *verbose=2*, *follow_state=False*)

Davidson diagonalization to solve the non-symmetric eigenvalue problem

Args:

aop [function([x]) => [array_like_x]] Matrix vector multiplication $y_{ki} = \sum_j a_{ij} * x_{jk}$.

x0 [1D array or a list of 1D array] Initial guess. The initial guess vector(s) are just used as the initial subspace bases. If the subspace is smaller than "nroots", eg 10 roots and one initial guess, all eigenvectors are chosen as the eigenvectors during the iterations. The first iteration has one eigenvector, the next iteration has two, the third iterastion has 4, ..., until the subspace size > nroots.

precond [function(dx, e, x0) => array_like_dx] Preconditioner to generate new trial vector. The argument dx is a residual vector `a*x0-e*x0`; e is the current eigenvalue; x0 is the current eigenvector.

Kwargs:

tol [float] Convergence tolerance.

max_cycle [int] max number of iterations.

max_space [int] space size to hold trial vectors.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

max_memory [int or float] Allowed memory in MB.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

nroots [int] Number of eigenvalues to be computed. When nroots > 1, it affects the shape of the return value

lessio [bool] How to compute a*x0 for current eigenvector x0. There are two ways to compute a*x0. One is to assemble the existed a*x. The other is to call aop(x0). The default is the first method which needs more IO and less computational cost. When IO is slow, the second method can be considered.

left [bool] Whether to calculate and return left eigenvectors. Default is False.

pick [function(w,v,nroots) => (e[idx], w[:,idx], idx)] Function to filter eigenvalues and eigenvectors.

follow_state [bool] If the solution dramatically changes in two iterations, clean the subspace and restart the iteration with the old solution. It can help to improve numerical stability. Default is False.

Returns:

conv [bool] Converged or not

e [list of eigenvalues] The eigenvalues can be sorted real or complex, depending on the return value of `pick` function.

vl [list of 1D arrays] Left eigenvectors. Only returned if `left=True`.

c [list of 1D arrays] Right eigenvectors.

Examples:

```
>>> from pyscf import lib
>>> a = numpy.random.random((10,10))
>>> a = a
>>> aop = lambda xs: [numpy.dot(a,x) for x in xs]
>>> precond = lambda dx, e, x0: dx/(a.diagonal()-e)
>>> x0 = a[0]
>>> e, vl, vr = lib.davidson(aop, x0, precond, nroots=2, left=True)
>>> len(e)
2
```

pyscf.lib.linalg_helper.**dgeev**(*abop*, *x0*, *precond*, *type=1*, *tol=1e-12*, *max_cycle=50*, *max_space=12*, *lindep=1e-14*, *max_memory=2000*, *dot=<built-in function dot>*, *callback=None*, *nroots=1*, *lessio=False*, *verbose=2*)

Davidson diagonalization method to solve A c = e B c.

Args:

abop [function(x) => (array_like_x, array_like_x)] abop applies two matrix vector multiplications and returns tuple (Ax, Bx)

x0 [1D array] Initial guess

precond [function(dx, e, x0) => array_like_dx] Preconditioner to generate new trial vector. The argument dx is a residual vector `a*x0-e*x0`; e is the current eigenvalue; x0 is the current eigenvector.

Kwargs:

tol [float] Convergence tolerance.

max_cycle [int] max number of iterations.

max_space [int] space size to hold trial vectors.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

max_memory [int or float] Allowed memory in MB.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

nroots [int] Number of eigenvalues to be computed. When nroots > 1, it affects the shape of the return value

lessio [bool] How to compute a*x0 for current eigenvector x0. There are two ways to compute a*x0. One is to assemble the existed a*x. The other is to call aop(x0). The default is the first method which needs more IO and less computational cost. When IO is slow, the second method can be considered.

Returns:

e [list of floats] The lowest `nroots` eigenvalues.

c [list of 1D arrays] The lowest `nroots` eigenvectors.

`pyscf.lib.linalg_helper.`**`dgeev1`**(*abop*, *x0*, *precond*, *type=1*, *tol=1e-12*, *max_cycle=50*, *max_space=12*, *lindep=1e-14*, *max_memory=2000*, *dot=<built-in function dot>*, *callback=None*, *nroots=1*, *lessio=False*, *verbose=2*)

Davidson diagonalization method to solve A c = e B c.

Args:

abop [function([x]) => ([array_like_x], [array_like_x])] abop applies two matrix vector multiplications and returns tuple (Ax, Bx)

x0 [1D array] Initial guess

precond [function(dx, e, x0) => array_like_dx] Preconditioner to generate new trial vector. The argument dx is a residual vector `a*x0-e*x0`; e is the current eigenvalue; x0 is the current eigenvector.

Kwargs:

tol [float] Convergence tolerance.

max_cycle [int] max number of iterations.

max_space [int] space size to hold trial vectors.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

max_memory [int or float] Allowed memory in MB.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

nroots [int] Number of eigenvalues to be computed. When nroots > 1, it affects the shape of the return value

lessio [bool] How to compute a*x0 for current eigenvector x0. There are two ways to compute a*x0. One is to assemble the existed a*x. The other is to call aop(x0). The default is the first method which needs more IO and less computational cost. When IO is slow, the second method can be considered.

Returns:

conv [bool] Converged or not

e [list of floats] The lowest `nroots` eigenvalues.

c [list of 1D arrays] The lowest `nroots` eigenvectors.

`pyscf.lib.linalg_helper.`**`dsolve`**(*aop*, *b*, *precond*, *tol=1e-12*, *max_cycle=30*, *dot=<built-in function dot>*, *lindep=1e-16*, *verbose=0*)

Davidson iteration to solve linear equation. It works bad.

`pyscf.lib.linalg_helper.`**`eig`**(*aop*, *x0*, *precond*, *tol=1e-12*, *max_cycle=50*, *max_space=12*, *lindep=1e-14*, *max_memory=2000*, *dot=<built-in function dot>*, *callback=None*, *nroots=1*, *lessio=False*, *left=False*, *pick=<function pick_real_eigs>*, *verbose=2*, *follow_state=False*)

Davidson diagonalization to solve the non-symmetric eigenvalue problem

Args:

aop [function([x]) => [array_like_x]] Matrix vector multiplication $y_{ki} = \sum_j a_{ij} * x_{jk}$.

x0 [1D array or a list of 1D array] Initial guess. The initial guess vector(s) are just used as the initial subspace bases. If the subspace is smaller than "nroots", eg 10 roots and one initial guess, all eigenvectors are chosen as the eigenvectors during the iterations. The first iteration has one eigenvector, the next iteration has two, the third iterastion has 4, ..., until the subspace size > nroots.

precond [function(dx, e, x0) => array_like_dx] Preconditioner to generate new trial vector. The argument dx is a residual vector `a*x0-e*x0`; e is the current eigenvalue; x0 is the current eigenvector.

Kwargs:

tol [float] Convergence tolerance.

max_cycle [int] max number of iterations.

max_space [int] space size to hold trial vectors.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

max_memory [int or float] Allowed memory in MB.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

nroots [int] Number of eigenvalues to be computed. When nroots > 1, it affects the shape of the return value

lessio [bool] How to compute a*x0 for current eigenvector x0. There are two ways to compute a*x0. One is to assemble the existed a*x. The other is to call aop(x0). The default is the first method which needs more IO and less computational cost. When IO is slow, the second method can be considered.

left [bool] Whether to calculate and return left eigenvectors. Default is False.

pick [function(w,v,nroots) => (e[idx], w[:,idx], idx)] Function to filter eigenvalues and eigenvectors.

follow_state [bool] If the solution dramatically changes in two iterations, clean the subspace and restart the iteration with the old solution. It can help to improve numerical stability. Default is False.

Returns:

conv [bool] Converged or not

e [list of eigenvalues] The eigenvalues can be sorted real or complex, depending on the return value of `pick` function.

vl [list of 1D arrays] Left eigenvectors. Only returned if `left=True`.

c [list of 1D arrays] Right eigenvectors.

Examples:

```
>>> from pyscf import lib
>>> a = numpy.random.random((10,10))
>>> a = a
>>> aop = lambda xs: [numpy.dot(a,x) for x in xs]
>>> precond = lambda dx, e, x0: dx/(a.diagonal()-e)
>>> x0 = a[0]
>>> e, vl, vr = lib.davidson(aop, x0, precond, nroots=2, left=True)
>>> len(e)
2
```

pyscf.lib.linalg_helper.**eigh_by_blocks**(*h*, *s=None*, *labels=None*)

Solve an ordinary or generalized eigenvalue problem for diagonal blocks. The diagonal blocks are extracted based on the given basis "labels". The rows and columns which have the same labels are put in the same block. One common scenario one needs the block-wise diagonalization is to diagonalize the matrix in symmetry adapted basis, in which "labels" is the irreps of each basis.

Args:

h, s [2D array] Complex Hermitian or real symmetric matrix.

Kwargs: labels : list

Returns: w, v. w is the eigenvalue vector; v is the eigenfunction array; seig is the eigenvalue vector of the metric s.

Examples:

```
>>> from pyscf import lib
>>> a = numpy.ones((4,4))
>>> a[0::3,0::3] = 0
>>> a[1::3,1::3] = 2
>>> a[2::3,2::3] = 4
>>> labels = ['a', 'b', 'c', 'a']
>>> lib.eigh_by_blocks(a, labels)
(array([ 0.,  0.,  2.,  4.]),
 array([[ 1.,  0.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.],
        [ 0.,  1.,  0.,  0.]]))
>>> numpy.linalg.eigh(a)
(array([ -8.82020545e-01,  -1.81556477e-16,   1.77653793e+00,   5.10548262e+00]),
 array([[  6.40734630e-01,  -7.07106781e-01,   1.68598330e-01,   -2.47050070e-01],
        [ -3.80616542e-01,   9.40505244e-17,   8.19944479e-01,   -4.27577008e-01],
        [ -1.84524565e-01,   9.40505244e-17,  -5.20423152e-01,   -8.33732828e-01],
        [  6.40734630e-01,   7.07106781e-01,   1.68598330e-01,   -2.47050070e-
↪01]]))
```

```
>>> from pyscf import gto, lib, symm
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz', symmetry=True)
>>> c = numpy.hstack(mol.symm_orb)
>>> vnuc_so = reduce(numpy.dot, (c.T, mol.intor('int1e_nuc_sph'), c))
>>> orbsym = symm.label_orb_symm(mol, mol.irrep_name, mol.symm_orb, c)
>>> lib.eigh_by_blocks(vnuc_so, labels=orbsym)
(array([-4.50766885, -1.80666351, -1.7808565 , -1.7808565 , -1.74189134,
        -0.98998583, -0.98998583, -0.40322226, -0.30242374, -0.07608981]),
 ...)
```

pyscf.lib.linalg_helper.**krylov**(*aop*, *b*, *x0=None*, *tol=1e-10*, *max_cycle=30*, *dot=<built-in function dot>*, *lindep=1e-15*, *callback=None*, *hermi=False*, *verbose=2*)

Krylov subspace method to solve (1+a) x = b. Ref: J. A. Pople et al, Int. J. Quantum. Chem. Symp. 13, 225 (1979).

Args:

aop [function(x) => array_like_x] aop(x) to mimic the matrix vector multiplication $\sum_j a_{ij} x_j$. The argument is a 1D array. The returned value is a 1D array.

Kwargs:

x0 [1D array] Initial guess

tol [float] Tolerance to terminate the operation aop(x).

max_cycle [int] max number of iterations.

lindep [float] Linear dependency threshold. The function is terminated when the smallest eigenvalue of the metric of the trial vectors is lower than this threshold.

dot [function(x, y) => scalar] Inner product

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

Returns: x : 1D array like b

Examples:

```
>>> from pyscf import lib
>>> a = numpy.random.random((10,10)) * 1e-2
>>> b = numpy.random.random(10)
>>> aop = lambda x: numpy.dot(a,x)
>>> x = lib.krylov(aop, b)
>>> numpy.allclose(numpy.dot(a,x)+x, b)
True
```

pyscf.lib.linalg_helper.**safe_eigh**(*h*, *s*, *lindep=1e-15*)

Solve generalized eigenvalue problem h v = w s v.

Note: The number of eigenvalues and eigenvectors might be less than the matrix dimension if linear dependency is found in metric s.

Args:

h, s [2D array] Complex Hermitian or real symmetric matrix.

Kwargs:

lindep [float] Linear dependency threshold. By diagonalizing the metric s, we consider the eigenvectors are linearly dependent subsets if their eigenvalues are smaller than this threshold.

Returns: w, v, seig. w is the eigenvalue vector; v is the eigenfunction array; seig is the eigenvalue vector of the metric s.

1.5.4 chkfile

pyscf.lib.chkfile.**dump**(*chkfile*, *key*, *value*)
Save array(s) in chkfile

Args:

chkfile [str] Name of chkfile.

key : str

value [array, vector ... or dict] If value is a python dict, the key/value of the dict will be saved recursively as the HDF5 group/dataset

Returns: No return value

Examples:

```
>>> import h5py
>>> from pyscf import lib
>>> ci = {'Ci' : {'op': ('E', 'i'), 'irrep': ('Ag', 'Au')}}
>>> lib.chkfile.save('symm.chk', 'symm', ci)
>>> f = h5py.File('symm.chk')
>>> f.keys()
['symm']
>>> f['symm'].keys()
['Ci']
>>> f['symm/Ci'].keys()
['op', 'irrep']
>>> f['symm/Ci/op']
<HDF5 dataset "op": shape (2,), type "|S1">
```

pyscf.lib.chkfile.**dump_mol**(*mol*, *chkfile*)
Save Mole object in chkfile

Args: mol : an instance of `Mole`.

chkfile [str] Name of chkfile.

Returns: No return value

pyscf.lib.chkfile.**load**(*chkfile*, *key*)
Load array(s) from chkfile

Args:

chkfile [str] Name of chkfile. The chkfile needs to be saved in HDF5 format.

key [str] HDF5.dataset name or group name. If key is the HDF5 group name, the group will be loaded into an Python dict, recursively

Returns: whatever read from chkfile

Examples:

```
>>> from pyscf import gto, scf, lib
>>> mol = gto.M(atom='He 0 0 0')
>>> mf = scf.RHF(mol)
>>> mf.chkfile = 'He.chk'
>>> mf.kernel()
>>> mo_coeff = lib.chkfile.load('He.chk', 'scf/mo_coeff')
>>> mo_coeff.shape
(1, 1)
>>> scfdat = lib.chkfile.load('He.chk', 'scf')
>>> scfdat.keys()
['e_tot', 'mo_occ', 'mo_energy', 'mo_coeff']
```

pyscf.lib.chkfile.**load_mol**(*chkfile*)

Load Mole object from chkfile. The save_mol/load_mol operation can be used a serialization method for Mole object.

Args:

chkfile [str] Name of chkfile.

Returns: A (initialized/built) Mole object

Examples:

```
>>> from pyscf import gto, lib
>>> mol = gto.M(atom='He 0 0 0')
>>> lib.chkfile.save_mol(mol, 'He.chk')
>>> lib.chkfile.load_mol('He.chk')
<pyscf.gto.mole.Mole object at 0x7fdcd94d7f50>
```

pyscf.lib.chkfile.**save_mol**(*mol*, *chkfile*)

Save Mole object in chkfile

Args: mol : an instance of `Mole`.

chkfile [str] Name of chkfile.

Returns: No return value

Fast load

The results of SCF and MCSCF methods are saved as a Python dictionary in the chkfile. One can fast load the results and update the SCF and MCSCF objects using the python built in methods `.__dict__.update`, eg:

```
from pyscf import gto, scf, mcscf, lib
mol = gto.M(atom='N 0 0 0; N 1 1 1', basis='ccpvdz')
mf = mol.apply(scf.RHF).set(chkfile='n2.chk').run()
mc = mcscf.CASSCF(mf, 6, 6).set(chkfile='n2.chk').run()

# load SCF results
mf = scf.RHF(mol)
mf.__dict__.update(lib.chkfile.load('n2.chk', 'scf'))

# load MCSCF results
mc = mcscf.CASCI(mf, 6, 6)
mc.__dict__.update(lib.chkfile.load('n2.chk', 'mcscf'))
mc.kernel()
```

1.6 scf — Mean-field methods

1.6.1 Stability analysis

1.6.2 Addons

Special treatments may be required to the SCF methods in some situations. These special treatments cannot be universally applied for all SCF models. They were defined in the `scf.addons` module. For example, in an UHF calculation, we may want the S_z value to be changed (the numbers of alpha and beta electrons not conserved) during SCF iteration while conserving the total number of electrons. `scf.addons.dynamic_sz_()` can provide this functionality:

```
from pyscf import gto, scf
mol = gto.M(atom='O 0 0 0; O 0 0 1')
mf = scf.UHF(mol)
mf.verbose=4
mf = scf.addons.dynamic_sz_(mf)
mf.kernel()
print('S^2 = %s, 2S+1 = %s' % mf.spin_square())
```

This function automatically converges the ground sate of oxygen molecule to triplet state although we didn't specify spin state in the `mol` object.

Note: Function `scf.addons.dynamic_sz_()` has side effects. It changes the underlying mean-field object.

The *addons* mechanism increases the flexibility of PySCf program. You can define various addons to customize the default behaviour of pyscf program. For example, if you'd like to track the changes of the density (the diagonal term of density matrix) of certain basis during the SCF iteration, you can write the following addon to output the required density:

```
def output_density(mf, basis_label):
    ao_labels = mf.mol.ao_labels()
    old_make_rdm1 = mf.make_rdm1
    def make_rdm1(mo_coeff, mo_occ):
        dm = old_make_rdm1(mo_coeff, mo_occ)
        print('AO          alpha             beta')
        for i,s in enumerate(ao_labels):
            if basis_label in s:
                print(s, dm[0][i,i], dm[1][i,i])
        return dm
    mf.make_rdm1 = make_rdm1
    return mf
from pyscf import gto, scf
mol = gto.M(atom='O 0 0 0; O 0 0 1')
mf = scf.UHF(mol)
mf.verbose=4
mf = scf.addons.dynamic_sz_(mf)
mf = output_density(mf, 'O 2p')
mf.kernel()
```

1.6.3 Caching two-electron integrals

When memory is enough (specified by the `max_memory` of SCF object), the SCF object generates all two-electron integrals in memory and cache them in `_eri` array. The default `max_memory` (defined in `lib.parameters.MAX_MEMORY`, see *Maximum memory*) is 4 GB. It roughly corresponds to two-electron real integrals for 250 orbitals. For small systems, the cached integrals usually provide the best performance. If you have enough main memory in your computer, you can increase the `max_memory` of SCF object to cache the integrals in memory.

The cached integrals `_eri` are treated as a dense tensor. When system becomes larger and the two-electron integral tensor becomes sparse, caching integrals may lose performance advantage. This is mainly due to the fact that the implementation of J/K build for the cached integrals did not utilize the sparsity of the integral tensor. Also, the data locality was not considered in the implementation which sometimes leads to bad OpenMP multi-threading speed up. For large system, the AO-driven direct SCF method is more favorable.

1.6.4 Customizing Hamiltonian

This integral object `_eri` is not merely used by the mean-field calculation. Along with the `get_hcore()` method, this two-electron integral object will be treated as the Hamiltonian in the post-SCF code whenever possible. This mechanism provides a way to model arbitrary fermion system in PySCF. You can customize a system by changing the 1-electron Hamiltonian and the mean-field `_eri` attribute. For example, the following code solves a model system:

```
import numpy
from pyscf import gto, scf, ao2mo, ccsd
mol = gto.M()
n = 10
mol.nelectron = 10
mf = scf.RHF(mol)

t = numpy.zeros((n,n))
for i in range(n-1):
    t[i,i+1] = t[i+1,i] = -1.0
t[n-1,0] = t[0,n-1] = 1.0  # anti-PBC
eri = numpy.zeros((n,n,n,n))
for i in range(n):
    eri[i,i,i,i] = 4.0

mf.get_hcore = lambda *args: t
mf.get_ovlp = lambda *args: numpy.eye(n)
# ao2mo.restore(8, eri, n) to get 8-fold symmetry of the integrals
# ._eri only supports the 2-electron integrals in 4-fold or 8-fold symmetry.
mf._eri = ao2mo.restore(8, eri, n)

mf.kernel()

mycc = ccsd.RCCSD(mf).run()
e,v = mycc.ipccsd(nroots=3)
print('IP = ', e)
e,v = mycc.eaccsd(nroots=3)
print('EA = ', e)
```

Some post-SCF methods require the 4-index MO integrals. Depending the available memory (affected by the value of `max_memory` in each class), these methods may not use the "AO integrals" cached in `_eri`. To ensure the post mean-field methods to use the `_eri` integrals no matter whether the actual memory usage is over the `max_memory` limite, you can set the flag `incore_anyway` in `Mole` class to `True` before calling the `kernel()` function of the

post-SCF methods. In the following example, without setting `incore_anyway=True`, the CCSD calculations will crash:

```
import numpy
from pyscf import gto, scf, ao2mo, ccsd
mol = gto.M()
n = 10
mol.nelectron = n
mol.max_memory = 0
mf = scf.RHF(mol)

t = numpy.zeros((n,n))
for i in range(n-1):
    t[i,i+1] = t[i+1,i] = 1.0
t[n-1,0] = t[0,n-1] = -1.0
eri = numpy.zeros((n,n,n,n))
for i in range(n):
    eri[i,i,i,i] = 4.0

mf.get_hcore = lambda *args: t
mf.get_ovlp = lambda *args: numpy.eye(n)
mf._eri = ao2mo.restore(8, eri, n)
mf.kernel()

mol.incore_anyway = True
mycc = ccsd.RCCSD(mf).run()
e,v = mycc.ipccsd(nroots=3)
print('IP = ', e)
e,v = mycc.eaccsd(nroots=3)
print('EA = ', e)
```

Holding the entire two-particle interactions matrix elements in memory often leads to high memory usage. In the SCF calculation, the memory usage can be optimized if `_eri` is sparse. The SCF iterations requires only the Fock matrix which in turn calls the J/K build function `SCF.get_jk()` to compute the Coulomb and HF-exchange matrix. Overwriting the `SCF.get_jk()` function can reduce the memory footprint of the SCF part in the above example:

```
import numpy
from pyscf import gto, scf, ao2mo, ccsd
mol = gto.M()
n = 10
mol.nelectron = n
mol.max_memory = 0
mf = scf.RHF(mol)

t = numpy.zeros((n,n))
for i in range(n-1):
    t[i,i+1] = t[i+1,i] = 1.0
t[n-1,0] = t[0,n-1] = -1.0

mf.get_hcore = lambda *args: t
mf.get_ovlp = lambda *args: numpy.eye(n)
def get_jk(mol, dm, *args):
    j = numpy.diag(dm.diagonal()) * 4.
    k = numpy.diag(dm.diagonal()) * 4.
    return j, k
mf.get_jk = get_jk
mf.kernel()
```

Another way to handle the two-particle interactions of large model system is to use the density fitting/Cholesky decomposed integrals. See also *Saving/Loading DF integral tensor*.

1.6.5 Program reference

Non-relativistic and relativistic Hartree-Fock

Simple usage:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1')
>>> mf = scf.RHF(mol).run()
```

`scf.RHF()` returns an instance of SCF class. There are some parameters to control the SCF method.

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

DIIS [class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None/False to turn off DIIS.

diis [bool] whether to do DIIS. Default is True.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 0.

level_shift_factor [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

nelec [(int,int), for UHF/ROHF class] freeze the number of (alpha,beta) electrons.

irrep_nelec [dict, for symmetry- RHF/ROHF/UHF class only] to indicate the number of electrons for each irreps. In RHF, give {'ir_name':int, ...} ; In ROHF/UHF, give {'ir_name':(int,int), ...} . It is effective when `Mole.symmetry` is set `True`.

auxbasis [str, for density fitting SCF only] Auxiliary basis for density fitting.

```
>>> mf = scf.density_fit(scf.UHF(mol))
>>> mf.scf()
```

Density fitting can be applied to all non-relativistic HF class.

with_ssss [bool, for Dirac-Hartree-Fock only] If False, ignore small component integrals (SS|SS). Default is True.

with_gaunt [bool, for Dirac-Hartree-Fock only] If False, ignore Gaunt interaction. Default is False.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy [] Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Non-relativistic Hartree-Fock

class `pyscf.scf.hf.`**SCF**(*mol*)

SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

analyze (*verbose=None*, ***kwargs*)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Diople moment.

as_scanner (*mf*)
Generating a scanner/solver for HF PES.

The returned solver is a function. This function requires one argument "mol" as input and returns total HF energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the SCF object (DIIS, conv_tol, max_memory etc) are automatically applied in the solver.

Note scanner has side effects. It may change many underlying objects (_scf, with_df, with_x2c, ...) during calculation.

Examples:

```
>>> from pyscf import gto, scf
>>> hf_scanner = scf.RHF(gto.Mole().set(verbose=0)).as_scanner()
>>> hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
-98.552190448277955
>>> hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
-98.414750424294368
```

canonicalize (*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix within occupied, open, virtual subspaces separatedly (without change occupancy).

dip_moment (*mol=None*, *dm=None*, *unit_symbol=None*, *verbose=3*)
Dipole moment calculation

$$\mu_x = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A$$
$$\mu_y = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A$$
$$\mu_z = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A$$

where μ_x, μ_y, μ_z are the x, y and z components of dipole moment

Args: mol: an instance of `Mole` dm : a 2D ndarrays density matrices

Return: A list: the dipole moment on x, y and z component

eig (*h*, *s*)
Solver for generalized eigenvalue problem

$$HC = SCE$$

energy_elec(*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Electronic part of Hartree-Fock energy, for given core hamiltonian and HF potential

... math:

```
E = \sum_{ij}h_{ij} \gamma_{ji}
  + \frac{1}{2}\sum_{ijkl} \gamma_{ji}\gamma_{lk} \langle ik||jl\rangle
```

Args: mf : an instance of SCF class

Kwargs:

dm [2D ndarray] one-partical density matrix

h1e [2D ndarray] Core hamiltonian

vhf [2D ndarray] HF potential

Returns: Hartree-Fock electronic energy and the Coulomb energy

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> dm = mf.make_rdm1()
>>> scf.hf.energy_elec(mf, dm)
(-1.5176090667746334, 0.60917167853723675)
```

energy_tot(*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Total Hartree-Fock energy, electronic part plus nuclear repulstion See `scf.hf.energy_elec()` for the electron part

from_chk(*chkfile=None*, *project=True*)
Read the HF results from checkpoint file, then project it to the basis defined by `mol`

Returns: Density matrix, 2D ndarray

get_fock(*mf*, *h1e=None*, *s1e=None*, *vhf=None*, *dm=None*, *cycle=-1*, *diis=None*, *diis_start_cycle=None*, *level_shift_factor=None*, *damp_factor=None*)
F = h^{core} + V^{HF}

Special treatment (damping, DIIS, or level shift) will be applied to the Fock matrix if diis and cycle is specified (The two parameters are passed to get_fock function during the SCF iteration)

Args:

h1e [2D ndarray] Core hamiltonian

s1e [2D ndarray] Overlap matrix, for DIIS

vhf [2D ndarray] HF potential matrix

dm [2D ndarray] Density matrix, for DIIS

Kwargs:

cycle [int] Then present SCF iteration step, for DIIS

diis [an object of `SCF.DIIS` class] DIIS object to hold intermediate Fock and error vectors

diis_start_cycle [int] The step to start DIIS. Default is 0.

level_shift_factor [float or int] Level shift (in AU) for virtual space. Default is 0.

get_grad(*mo_coeff*, *mo_occ*, *fock=None*)
RHF Gradients

Args:

mo_coeff [2D ndarray] Obital coefficients

mo_occ [1D ndarray] Orbital occupancy

fock_ao [2D ndarray] Fock matrix in AO representation

Returns: Gradients in MO representation. It's a num_occ*num_vir vector.

get_j(*mol=None*, *dm=None*, *hermi=1*)
Compute J matrix for the given density matrix.

get_jk(*mol=None*, *dm=None*, *hermi=1*)
Compute J, K matrices for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

Examples:

```
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dms = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> j, k = scf.hf.get_jk(mol, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

get_k(*mol=None*, *dm=None*, *hermi=1*)
Compute K matrix for the given density matrix.

get_occ(*mf*, *mo_energy=None*, *mo_coeff=None*)
Label the occupancies for each orbital

Kwargs:

mo_energy [1D ndarray] Obital energies

mo_coeff [2D ndarray] Obital coefficients

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> energy = numpy.array([-10., -1., 1, -2., 0, -3])
>>> mf.get_occ(energy)
array([2, 2, 0, 2, 2, 2])
```

get_veff(*mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Hartree-Fock potential matrix for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. If not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: matrix Vhf = 2*J - K. Vhf can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dm0 = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> vhf0 = scf.hf.get_veff(mol, dm0, hermi=0)
>>> dm1 = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> vhf1 = scf.hf.get_veff(mol, dm1, hermi=0)
>>> vhf2 = scf.hf.get_veff(mol, dm1, dm_last=dm0, vhf_last=vhf0, hermi=0)
>>> numpy.allclose(vhf1, vhf2)
True
```

init_guess_by_1e(*mol=None*)
Generate initial guess density matrix from core hamiltonian

Returns: Density matrix, 2D ndarray

init_guess_by_atom(*mol=None*)
Generate initial guess density matrix from superposition of atomic HF density matrix. The atomic HF is occupancy averaged RHF

Returns: Density matrix, 2D ndarray

init_guess_by_chkfile(*chkfile=None*, *project=True*)
Read the HF results from checkpoint file, then project it to the basis defined by `mol`

Returns: Density matrix, 2D ndarray

init_guess_by_minao(*mol=None*)

Generate initial guess density matrix based on ANO basis, then project the density matrix to the basis set defined by `mol`

Returns: Density matrix, 2D ndarray

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> scf.hf.init_guess_by_minao(mol)
array([[ 0.94758917,  0.09227308],
       [ 0.09227308,  0.94758917]])
```

kernel(*dm0=None*)

main routine for SCF

Kwargs:

dm0 [ndarray] If given, it will be used as the initial guess density matrix

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> dm_guess = numpy.eye(mol.nao_nr())
>>> mf.kernel(dm_guess)
converged SCF energy = -98.5521904482821
-98.552190448282104
```

make_rdm1(*mo_coeff=None*, *mo_occ=None*)

One-particle density matrix in AO representation

Args:

mo_coeff [2D ndarray] Orbital coefficients. Each column is one orbital.

mo_occ [1D ndarray] Occupancy

mulliken_meta(*mol=None*, *dm=None*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)

Mulliken population analysis, based on meta-Lowdin AOs. In the meta-lowdin, the AOs are grouped in three sets: core, valence and Rydberg, the orthogonalization are carreid out within each subsets.

Args: mol : an instance of `Mole`

dm [ndarray or 2-item list of ndarray] Density matrix. ROHF dm is a 2-item list of 2D array

Kwargs: verbose : int or instance of `lib.logger.Logger`

pre_orth_method [str] Pre-orthogonalization, which localized GTOs for each atom. To obtain the occupied and unoccupied atomic shells, there are three methods

'ano' : Project GTOs to ANO basis
'minao' : Project GTOs to MINAO basis
'scf' : Fraction-averaged RHF

mulliken_pop(*mol=None*, *dm=None*, *s=None*, *verbose=5*)
Mulliken population analysis

$$M_{ij} = D_{ij}S_{ji}$$

Mulliken charges

$$\delta_i = \sum_j M_{ij}$$

pop(**args*, ***kwargs*)
Mulliken population analysis, based on meta-Lowdin AOs. In the meta-lowdin, the AOs are grouped in three sets: core, valence and Rydberg, the orthogonalization are carreid out within each subsets.

Args: mol : an instance of `Mole`

dm [ndarray or 2-item list of ndarray] Density matrix. ROHF dm is a 2-item list of 2D array

Kwargs: verbose : int or instance of `lib.logger.Logger`

pre_orth_method [str] Pre-orthogonalization, which localized GTOs for each atom. To obtain the occupied and unoccupied atomic shells, there are three methods

'ano' : Project GTOs to ANO basis
'minao' : Project GTOs to MINAO basis
'scf' : Fraction-averaged RHF

scf(*dm0=None*)
main routine for SCF

Kwargs:

dm0 [ndarray] If given, it will be used as the initial guess density matrix

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> dm_guess = numpy.eye(mol.nao_nr())
>>> mf.kernel(dm_guess)
converged SCF energy = -98.5521904482821
-98.552190448282104
```

update(*chkfile=None*)
Read attributes from the chkfile then replace the attributes of current object. See also mf.update_from_chk

---

class `pyscf.scf.hf.`**RHF**(*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

convert_from_(*mf*)
Convert given mean-field object to RHF/ROHF

get_jk(*mol=None*, *dm=None*, *hermi=1*)
Compute J, K matrices for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

Examples:

```
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dms = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> j, k = scf.hf.get_jk(mol, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

---

class pyscf.scf.rohf.**ROHF**(*mol*)

SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

analyze (*verbose=None*, ***kwargs*)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis

canonicalize (*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix within occupied, open, virtual subspaces separatedly (without change occupancy).

eig (*fock*, *s*)
Solver for generalized eigenvalue problem

$$HC = SCE$$

get_fock (*mf*, *h1e=None*, *s1e=None*, *vhf=None*, *dm=None*, *cycle=-1*, *diis=None*, *diis_start_cycle=None*, *level_shift_factor=None*, *damp_factor=None*)
Build fock matrix based on Roothaan's effective fock. See also `get_roothaan_fock()`

get_grad (*mo_coeff*, *mo_occ*, *fock=None*)
ROHF gradients is the off-diagonal block [co + cv + ov], where [cc co cv] [oc oo ov] [vc vo vv]

get_occ (*mf*, *mo_energy=None*, *mo_coeff=None*)
Label the occupancies for each orbital. NOTE the occupancies are not assigned based on the orbital energy ordering. The first N orbitals are assigned to be occupied orbitals.

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; O 0 0 1.1', spin=1)
>>> mf = scf.hf.SCF(mol)
>>> energy = numpy.array([-10., -1., 1, -2., 0, -3])
>>> mf.get_occ(energy)
array([2, 2, 2, 2, 1, 0])
```

get_veff (*mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Unrestricted Hartree-Fock potential matrix of alpha and beta spins, for the given density matrix

$$V_{ij}^{\alpha} = \sum_{kl}(ij|kl)(\gamma_{lk}^{\alpha} + \gamma_{lk}^{\beta}) - \sum_{kl}(il|kj)\gamma_{lk}^{\alpha}$$
$$V_{ij}^{\beta} = \sum_{kl}(ij|kl)(\gamma_{lk}^{\alpha} + \gamma_{lk}^{\beta}) - \sum_{kl}(il|kj)\gamma_{lk}^{\beta}$$

Args: mol : an instance of `Mole`

dm [a list of ndarrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. When it is not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: $V_{hf} = (V^{\alpha}, V^{\beta})$. V^{α} (and V^{β}) can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dmsa = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dmsb = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dms = numpy.vstack((dmsa,dmsb))
>>> dms.shape
(6, 2, 2)
>>> vhfa, vhfb = scf.uhf.get_veff(mol, dms, hermi=0)
>>> vhfa.shape
(3, 2, 2)
>>> vhfb.shape
(3, 2, 2)
```

make_rdm1(*mo_coeff=None*, *mo_occ=None*)
One-particle densit matrix. mo_occ is a 1D array, with occupancy 1 or 2.

---

class pyscf.scf.uhf.**UHF**(*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for UHF:

nelec [(int, int)] If given, freeze the number of (alpha,beta) electrons to the given value.

level_shift [number or two-element list] level shift (in Eh) for alpha and beta Fock if two-element list is given.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> print('S^2 = %.7f, 2S+1 = %.7f' % mf.spin_square())
S^2 = 0.7570150, 2S+1 = 2.0070027
```

canonicalize(*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separatedly (without change occupancy).

det_ovlp(*mo1*, *mo2*, *occ1*, *occ2*, *ovlp=None*)
Calculate the overlap between two different determinants. It is the product of single values of molecular

orbital overlap matrix.

$$S_{12} = \langle \Psi_A | \Psi_B \rangle = (\det \mathbf{U})(\det \mathbf{V}^\dagger) \prod_{i=1}^{2N} \lambda_{ii}$$

where $\mathbf{U}, \mathbf{V}, \lambda$ are unitary matrices and single values generated by single value decomposition(SVD) of the overlap matrix $\mathbf{O}$ which is the overlap matrix of two sets of molecular orbitals:

$$\mathbf{U}^\dagger \mathbf{O} \mathbf{V} = \mathbf{\Lambda}$$

Args:

mo1, mo2 [2D ndarrays] Molecualr orbital coefficients

occ1, occ2: 2D ndarrays occupation numbers

Return:

A list: the product of single values: float x_a, x_b: 1D ndarrays $\mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{V}^\dagger$ They are used to calculate asymmetric density matrix

dip_moment (*mol=None*, *dm=None*, *unit_symbol=None*, *verbose=3*)
Dipole moment calculation

$$\mu_x = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A$$
$$\mu_y = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A$$
$$\mu_z = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A$$

where μ_x, μ_y, μ_z are the x, y and z components of dipole moment

Args: mol: an instance of `Mole`

dm [a list of 2D ndarrays] a list of density matrices

Return: A list: the dipole moment on x, y and z component

energy_elec (*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Electronic energy of Unrestricted Hartree-Fock

Returns: Hartree-Fock electronic energy and the 2-electron part contribution

get_jk (*mol=None*, *dm=None*, *hermi=1*)
Coulomb (J) and exchange (K)

Args:

dm [a list of 2D arrays or a list of 3D arrays] (alpha_dm, beta_dm) or (alpha_dms, beta_dms)

get_veff (*mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Unrestricted Hartree-Fock potential matrix of alpha and beta spins, for the given density matrix

$$V_{ij}^\alpha = \sum_{kl}(ij|kl)(\gamma_{lk}^\alpha + \gamma_{lk}^\beta) - \sum_{kl}(il|kj)\gamma_{lk}^\alpha$$
$$V_{ij}^\beta = \sum_{kl}(ij|kl)(\gamma_{lk}^\alpha + \gamma_{lk}^\beta) - \sum_{kl}(il|kj)\gamma_{lk}^\beta$$

Args: mol : an instance of `Mole`

dm [a list of ndarrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. When it is not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: $V_{hf} = (V^{\alpha}, V^{\beta})$. V^{α} (and V^{β}) can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dmsa = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dmsb = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dms = numpy.vstack((dmsa,dmsb))
>>> dms.shape
(6, 2, 2)
>>> vhfa, vhfb = scf.uhf.get_veff(mol, dms, hermi=0)
>>> vhfa.shape
(3, 2, 2)
>>> vhfb.shape
(3, 2, 2)
```

init_guess_by_minao(*mol=None*, *breaksym=True*)
Initial guess in terms of the overlap to minimal basis.

make_asym_dm(*mo1*, *mo2*, *occ1*, *occ2*, *x*)
One-particle asymmetric density matrix

Args:

mo1, mo2 [2D ndarrays] Molecualr orbital coefficients

occ1, occ2: 2D ndarrays Occupation numbers

x: 2D ndarrays $\mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{V}^{\dagger}$. See also `det_ovlp()`

Return: A list of 2D ndarrays for alpha and beta spin

Examples:

```
>>> mf1 = scf.UHF(gto.M(atom='H 0 0 0; F 0 0 1.3', basis='ccpvdz')).run()
>>> mf2 = scf.UHF(gto.M(atom='H 0 0 0; F 0 0 1.4', basis='ccpvdz')).run()
>>> s = gto.intor_cross('int1e_ovlp_sph', mf1.mol, mf2.mol)
>>> det, x = det_ovlp(mf1.mo_coeff, mf1.mo_occ, mf2.mo_coeff, mf2.mo_occ, s)
>>> adm = make_asym_dm(mf1.mo_coeff, mf1.mo_occ, mf2.mo_coeff, mf2.mo_occ, x)
>>> adm.shape
(2, 19, 19)
```

make_rdm1(*mo_coeff=None*, *mo_occ=None*)
One-particle density matrix

Returns: A list of 2D ndarrays for alpha and beta spins

spin_square(*mo_coeff=None*, *s=None*)
Spin of the given UHF orbitals

$$S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2$$

where $S_+ = \sum_i S_{i+}$ is effective for all beta occupied orbitals; $S_- = \sum_i S_{i-}$ is effective for all alpha occupied orbitals.

1.**There are two possibilities for** S_+S_-

(a) same electron $S_+S_- = \sum_i s_{i+}s_{i-}$,

$$\sum_i \langle UHF|s_{i+}s_{i-}|UHF\rangle = \sum_{pq}\langle p|s_+s_-|q\rangle\gamma_{qp} = n_\alpha$$

2) different electrons $S_+S_- = \sum s_{i+}s_{j-}, (i \neq j)$. There are in total $n(n-1)$ terms. As a two-particle operator,

$$\langle S_+S_-\rangle = \langle ij|s_+s_-|ij\rangle - \langle ij|s_+s_-|ji\rangle = -\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle$$

2.**Similarly, for** S_-S_+

(a) same electron

$$\sum_i \langle s_{i-}s_{i+}\rangle = n_\beta$$

(a) different electrons

$$\langle S_-S_+\rangle = -\langle i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta\rangle$$

2.**For** S_z^2

(a) same electron

$$\langle s_z^2\rangle = \frac{1}{4}(n_\alpha + n_\beta)$$

(a) different electrons

$$\begin{aligned}
&\frac{1}{2}\sum_{ij}(\langle ij|2s_{z1}s_{z2}|ij\rangle - \langle ij|2s_{z1}s_{z2}|ji\rangle) \\
&= \frac{1}{4}(\langle i^\alpha|i^\alpha\rangle\langle j^\alpha|j^\alpha\rangle - \langle i^\alpha|i^\alpha\rangle\langle j^\beta|j^\beta\rangle - \langle i^\beta|i^\beta\rangle\langle j^\alpha|j^\alpha\rangle + \langle i^\beta|i^\beta\rangle\langle j^\beta|j^\beta\rangle) \\
&- \frac{1}{4}(\langle i^\alpha|j^\alpha\rangle\langle j^\alpha|i^\alpha\rangle + \langle i^\beta|j^\beta\rangle\langle j^\beta|i^\beta\rangle) \\
&= \frac{1}{4}(n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2) - \frac{1}{4}(n_\alpha + n_\beta) \\
&= \frac{1}{4}((n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta))
\end{aligned}$$

In total

$$\langle S^2\rangle = \frac{1}{2}(n_\alpha - \sum_{ij}\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle + n_\beta - \sum_{ij}\langle i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta\rangle) + \frac{1}{4}(n_\alpha - n_\beta)^2$$

Args:

mo [a list of 2 ndarrays] Occupied alpha and occupied beta orbitals

Kwargs:

s [ndarray] AO overlap

Returns: A list of two floats. The first is the expectation value of S^2. The second is the corresponding 2S+1

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> mo = (mf.mo_coeff[0][:,mf.mo_occ[0]>0], mf.mo_coeff[1][:,mf.mo_occ[1]>0])
>>> print('S^2 = %.7f, 2S+1 = %.7f' % spin_square(mo, mol.intor('int1e_ovlp_
↪sph')))
S^2 = 0.7570150, 2S+1 = 2.0070027
```

Hartree-Fock

class `pyscf.scf.hf.`**RHF**(*mol*)

SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

convert_from_(*mf*)
Convert given mean-field object to RHF/ROHF

get_jk(*mol=None*, *dm=None*, *hermi=1*)
Compute J, K matrices for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

Examples:

```
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dms = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> j, k = scf.hf.get_jk(mol, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

class `pyscf.scf.hf.`**SCF**(*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

analyze(*verbose=None*, ***kwargs*)

Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Diople moment.

as_scanner(*mf*)

Generating a scanner/solver for HF PES.

The returned solver is a function. This function requires one argument "mol" as input and returns total HF energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the SCF object (DIIS, conv_tol, max_memory etc) are automatically applied in the solver.

Note scanner has side effects. It may change many underlying objects (_scf, with_df, with_x2c, ...) during calculation.

Examples:

```
>>> from pyscf import gto, scf
>>> hf_scanner = scf.RHF(gto.Mole().set(verbose=0)).as_scanner()
>>> hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
-98.552190448277955
>>> hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
-98.414750424294368
```

canonicalize (*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix within occupied, open, virtual subspaces separatedly (without change occupancy).

dip_moment (*mol=None*, *dm=None*, *unit_symbol=None*, *verbose=3*)
Dipole moment calculation

$$\mu_x = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A$$
$$\mu_y = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A$$
$$\mu_z = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A$$

where μ_x, μ_y, μ_z are the x, y and z components of dipole moment

Args: mol: an instance of `Mole` dm : a 2D ndarrays density matrices

Return: A list: the dipole moment on x, y and z component

eig (*h*, *s*)
Solver for generalized eigenvalue problem

$$HC = SCE$$

energy_elec (*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Electronic part of Hartree-Fock energy, for given core hamiltonian and HF potential

... math:

```
E = \sum_{ij}h_{ij} \gamma_{ji}
  + \frac{1}{2}\sum_{ijkl} \gamma_{ji}\gamma_{lk} \langle ik||jl\rangle
```

Args: mf : an instance of SCF class

Kwargs:

dm [2D ndarray] one-partical density matrix

h1e [2D ndarray] Core hamiltonian

vhf [2D ndarray] HF potential

Returns: Hartree-Fock electronic energy and the Coulomb energy

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> dm = mf.make_rdm1()
>>> scf.hf.energy_elec(mf, dm)
(-1.5176090667746334, 0.60917167853723675)
```

energy_tot (*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Total Hartree-Fock energy, electronic part plus nuclear repulstion See `scf.hf.energy_elec()` for the electron part

from_chk (*chkfile=None*, *project=True*)
Read the HF results from checkpoint file, then project it to the basis defined by `mol`

Returns: Density matrix, 2D ndarray

get_fock (*mf*, *h1e=None*, *s1e=None*, *vhf=None*, *dm=None*, *cycle=-1*, *diis=None*, *diis_start_cycle=None*, *level_shift_factor=None*, *damp_factor=None*)
F = h^{core} + V^{HF}

Special treatment (damping, DIIS, or level shift) will be applied to the Fock matrix if diis and cycle is specified (The two parameters are passed to get_fock function during the SCF iteration)

Args:

h1e [2D ndarray] Core hamiltonian

s1e [2D ndarray] Overlap matrix, for DIIS

vhf [2D ndarray] HF potential matrix

dm [2D ndarray] Density matrix, for DIIS

Kwargs:

cycle [int] Then present SCF iteration step, for DIIS

diis [an object of `SCF.DIIS` class] DIIS object to hold intermediate Fock and error vectors

diis_start_cycle [int] The step to start DIIS. Default is 0.

level_shift_factor [float or int] Level shift (in AU) for virtual space. Default is 0.

get_grad (*mo_coeff*, *mo_occ*, *fock=None*)
RHF Gradients

Args:

mo_coeff [2D ndarray] Obital coefficients

mo_occ [1D ndarray] Orbital occupancy

fock_ao [2D ndarray] Fock matrix in AO representation

Returns: Gradients in MO representation. It's a num_occ*num_vir vector.

get_j (*mol=None*, *dm=None*, *hermi=1*)
Compute J matrix for the given density matrix.

get_jk (*mol=None*, *dm=None*, *hermi=1*)
Compute J, K matrices for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

Examples:

```
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dms = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> j, k = scf.hf.get_jk(mol, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

get_k(*mol=None*, *dm=None*, *hermi=1*)
Compute K matrix for the given density matrix.

get_occ(*mf*, *mo_energy=None*, *mo_coeff=None*)
Label the occupancies for each orbital

Kwargs:

mo_energy [1D ndarray] Obital energies

mo_coeff [2D ndarray] Obital coefficients

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> energy = numpy.array([-10., -1., 1, -2., 0, -3])
>>> mf.get_occ(energy)
array([2, 2, 0, 2, 2, 2])
```

get_veff(*mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Hartree-Fock potential matrix for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. If not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: matrix Vhf = 2*J - K. Vhf can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dm0 = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> vhf0 = scf.hf.get_veff(mol, dm0, hermi=0)
>>> dm1 = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> vhf1 = scf.hf.get_veff(mol, dm1, hermi=0)
>>> vhf2 = scf.hf.get_veff(mol, dm1, dm_last=dm0, vhf_last=vhf0, hermi=0)
>>> numpy.allclose(vhf1, vhf2)
True
```

init_guess_by_1e(*mol=None*)
Generate initial guess density matrix from core hamiltonian

Returns: Density matrix, 2D ndarray

init_guess_by_atom(*mol=None*)
Generate initial guess density matrix from superposition of atomic HF density matrix. The atomic HF is occupancy averaged RHF

Returns: Density matrix, 2D ndarray

init_guess_by_chkfile(*chkfile=None*, *project=True*)
Read the HF results from checkpoint file, then project it to the basis defined by `mol`

Returns: Density matrix, 2D ndarray

init_guess_by_minao(*mol=None*)
Generate initial guess density matrix based on ANO basis, then project the density matrix to the basis set defined by `mol`

Returns: Density matrix, 2D ndarray

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> scf.hf.init_guess_by_minao(mol)
array([[ 0.94758917,  0.09227308],
       [ 0.09227308,  0.94758917]])
```

kernel(*dm0=None*)
main routine for SCF

Kwargs:

dm0 [ndarray] If given, it will be used as the initial guess density matrix

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> dm_guess = numpy.eye(mol.nao_nr())
>>> mf.kernel(dm_guess)
converged SCF energy = -98.5521904482821
-98.552190448282104
```

make_rdm1 (*mo_coeff=None*, *mo_occ=None*)
One-particle density matrix in AO representation

Args:

mo_coeff [2D ndarray] Orbital coefficients. Each column is one orbital.

mo_occ [1D ndarray] Occupancy

mulliken_meta (*mol=None*, *dm=None*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)
Mulliken population analysis, based on meta-Lowdin AOs. In the meta-lowdin, the AOs are grouped in three sets: core, valence and Rydberg, the orthogonalization are carreid out within each subsets.

Args: mol : an instance of `Mole`

dm [ndarray or 2-item list of ndarray] Density matrix. ROHF dm is a 2-item list of 2D array

Kwargs: verbose : int or instance of `lib.logger.Logger`

pre_orth_method [str] Pre-orthogonalization, which localized GTOs for each atom. To obtain the occupied and unoccupied atomic shells, there are three methods

'ano' : Project GTOs to ANO basis
'minao' : Project GTOs to MINAO basis
'scf' : Fraction-averaged RHF

mulliken_pop (*mol=None*, *dm=None*, *s=None*, *verbose=5*)
Mulliken population analysis

$$M_{ij} = D_{ij}S_{ji}$$

Mulliken charges

$$\delta_i = \sum_j M_{ij}$$

pop (**args*, ***kwargs*)
Mulliken population analysis, based on meta-Lowdin AOs. In the meta-lowdin, the AOs are grouped in three sets: core, valence and Rydberg, the orthogonalization are carreid out within each subsets.

Args: mol : an instance of `Mole`

dm [ndarray or 2-item list of ndarray] Density matrix. ROHF dm is a 2-item list of 2D array

Kwargs: verbose : int or instance of `lib.logger.Logger`

pre_orth_method [str] Pre-orthogonalization, which localized GTOs for each atom. To obtain the occupied and unoccupied atomic shells, there are three methods

'ano' : Project GTOs to ANO basis
'minao' : Project GTOs to MINAO basis
'scf' : Fraction-averaged RHF

scf (*dm0=None*)
main routine for SCF

Kwargs:

dm0 [ndarray] If given, it will be used as the initial guess density matrix

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> dm_guess = numpy.eye(mol.nao_nr())
>>> mf.kernel(dm_guess)
converged SCF energy = -98.5521904482821
-98.552190448282104
```

update (*chkfile=None*)
Read attributes from the chkfile then replace the attributes of current object. See also mf.update_from_chk

pyscf.scf.hf.**analyze** (*mf*, *verbose=5*, ***kwargs*)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Diople moment.

pyscf.scf.hf.**as_scanner** (*mf*)
Generating a scanner/solver for HF PES.

The returned solver is a function. This function requires one argument "mol" as input and returns total HF energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the SCF object (DIIS, conv_tol, max_memory etc) are automatically applied in the solver.

Note scanner has side effects. It may change many underlying objects (_scf, with_df, with_x2c, ...) during calculation.

Examples:

```
>>> from pyscf import gto, scf
>>> hf_scanner = scf.RHF(gto.Mole().set(verbose=0)).as_scanner()
>>> hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
-98.552190448277955
>>> hf_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
-98.414750424294368
```

pyscf.scf.hf.**canonicalize** (*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix within occupied, open, virtual subspaces separatedly (without change occupancy).

pyscf.scf.hf.**dip_moment** (*mol*, *dm*, *unit_symbol='Debye'*, *verbose=3*)

Dipole moment calculation

$$\mu_x = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A$$
$$\mu_y = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A$$
$$\mu_z = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A$$

where μ_x, μ_y, μ_z are the x, y and z components of dipole moment

Args: mol: an instance of `Mole` dm : a 2D ndarrays density matrices

Return: A list: the dipole moment on x, y and z component

`pyscf.scf.hf.`**`dot_eri_dm`**(*eri*, *dm*, *hermi=0*)

Compute J, K matrices in terms of the given 2-electron integrals and density matrix:

J ~ numpy.einsum('pqrs,qp->rs', eri, dm) K ~ numpy.einsum('pqrs,qr->ps', eri, dm)

Args:

eri [ndarray] 8-fold or 4-fold ERIs

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

Examples:

```
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> eri = _vhf.int2e_sph(mol._atm, mol._bas, mol._env)
>>> dms = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> j, k = scf.hf.dot_eri_dm(eri, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

`pyscf.scf.hf.`**`eig`**(*h*, *s*)

Solver for generalized eigenvalue problem

$$HC = SCE$$

`pyscf.scf.hf.`**`energy_elec`**(*mf*, *dm=None*, *h1e=None*, *vhf=None*)

Electronic part of Hartree-Fock energy, for given core hamiltonian and HF potential

... math:

```
E = \sum_{ij}h_{ij} \gamma_{ji}
  + \frac{1}{2}\sum_{ijkl} \gamma_{ji}\gamma_{lk} \langle ik||jl\rangle
```

Args: mf : an instance of SCF class

Kwargs:

dm [2D ndarray] one-partical density matrix

h1e [2D ndarray] Core hamiltonian

vhf [2D ndarray] HF potential

Returns: Hartree-Fock electronic energy and the Coulomb energy

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> dm = mf.make_rdm1()
>>> scf.hf.energy_elec(mf, dm)
(-1.5176090667746334, 0.60917167853723675)
```

pyscf.scf.hf.**energy_tot**(*mf*, *dm=None*, *h1e=None*, *vhf=None*)

Total Hartree-Fock energy, electronic part plus nuclear repulstion See `scf.hf.energy_elec()` for the electron part

pyscf.scf.hf.**get_fock**(*mf*, *h1e=None*, *s1e=None*, *vhf=None*, *dm=None*, *cycle=-1*, *diis=None*, *diis_start_cycle=None*, *level_shift_factor=None*, *damp_factor=None*)

F = h^{core} + V^{HF}

Special treatment (damping, DIIS, or level shift) will be applied to the Fock matrix if diis and cycle is specified (The two parameters are passed to get_fock function during the SCF iteration)

Args:

h1e [2D ndarray] Core hamiltonian

s1e [2D ndarray] Overlap matrix, for DIIS

vhf [2D ndarray] HF potential matrix

dm [2D ndarray] Density matrix, for DIIS

Kwargs:

cycle [int] Then present SCF iteration step, for DIIS

diis [an object of `SCF.DIIS` class] DIIS object to hold intermediate Fock and error vectors

diis_start_cycle [int] The step to start DIIS. Default is 0.

level_shift_factor [float or int] Level shift (in AU) for virtual space. Default is 0.

pyscf.scf.hf.**get_grad**(*mo_coeff*, *mo_occ*, *fock_ao*)

RHF Gradients

Args:

mo_coeff [2D ndarray] Obital coefficients

mo_occ [1D ndarray] Orbital occupancy

fock_ao [2D ndarray] Fock matrix in AO representation

Returns: Gradients in MO representation. It's a num_occ*num_vir vector.

pyscf.scf.hf.**get_hcore**(*mol*)
Core Hamiltonian

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> scf.hf.get_hcore(mol)
array([[-0.93767904, -0.59316327],
       [-0.59316327, -0.93767904]])
```

pyscf.scf.hf.**get_init_guess**(*mol*, *key='minao'*)
Pick a init_guess method

Kwargs:

key [str] One of 'minao', 'atom', '1e', 'chkfile'.

pyscf.scf.hf.**get_jk**(*mol*, *dm*, *hermi=1*, *vhfopt=None*)
Compute J, K matrices for the given density matrix

Args: mol : an instance of `Mole`

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

Examples:

```
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dms = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> j, k = scf.hf.get_jk(mol, dms, hermi=0)
>>> print(j.shape)
(3, 2, 2)
```

pyscf.scf.hf.**get_occ**(*mf*, *mo_energy=None*, *mo_coeff=None*)
Label the occupancies for each orbital

Kwargs:

mo_energy [1D ndarray] Obital energies

mo_coeff [2D ndarray] Obital coefficients

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1.1')
>>> mf = scf.hf.SCF(mol)
>>> energy = numpy.array([-10., -1., 1, -2., 0, -3])
>>> mf.get_occ(energy)
array([2, 2, 0, 2, 2, 2])
```

pyscf.scf.hf.**get_ovlp**(*mol*)
Overlap matrix

pyscf.scf.hf.**get_veff**(*mol*, *dm*, *dm_last=None*, *vhf_last=None*, *hermi=1*, *vhfopt=None*)
Hartree-Fock potential matrix for the given density matrix

Args: mol : an instance of Mole

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. If not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: matrix Vhf = 2*J - K. Vhf can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> from pyscf.scf import _vhf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dm0 = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> vhf0 = scf.hf.get_veff(mol, dm0, hermi=0)
>>> dm1 = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> vhf1 = scf.hf.get_veff(mol, dm1, hermi=0)
>>> vhf2 = scf.hf.get_veff(mol, dm1, dm_last=dm0, vhf_last=vhf0, hermi=0)
>>> numpy.allclose(vhf1, vhf2)
True
```

pyscf.scf.hf.**init_guess_by_1e**(*mol*)
Generate initial guess density matrix from core hamiltonian

Returns: Density matrix, 2D ndarray

pyscf.scf.hf.**init_guess_by_atom**(*mol*)
Generate initial guess density matrix from superposition of atomic HF density matrix. The atomic HF is occupancy averaged RHF

Returns: Density matrix, 2D ndarray

pyscf.scf.hf.**init_guess_by_chkfile**(*mol*, *chkfile_name*, *project=True*)
Read the HF results from checkpoint file, then project it to the basis defined by `mol`

Returns: Density matrix, 2D ndarray

pyscf.scf.hf.**init_guess_by_minao**(*mol*)
Generate initial guess density matrix based on ANO basis, then project the density matrix to the basis set defined by `mol`

Returns: Density matrix, 2D ndarray

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> scf.hf.init_guess_by_minao(mol)
array([[ 0.94758917,  0.09227308],
       [ 0.09227308,  0.94758917]])
```

pyscf.scf.hf.**kernel**(*mf*, *conv_tol=1e-10*, *conv_tol_grad=None*, *dump_chk=True*, *dm0=None*, *callback=None*, *conv_check=True*, ***kwargs*)
kernel: the SCF driver.

Args:

mf [an instance of SCF class] To hold the flags to control SCF. Besides the control parameters, one can modify its function members to change the behavior of SCF. The member functions which are called in kernel are

mf.get_init_guess
mf.get_hcore
mf.get_ovlp
mf.get_fock
mf.get_grad
mf.eig
mf.get_occ
mf.make_rdm1
mf.energy_tot
mf.dump_chk

Kwargs:

conv_tol [float] converge threshold.

conv_tol_grad [float] gradients converge threshold.

dump_chk [bool] Whether to save SCF intermediate results in the checkpoint file

dm0 [ndarray] Initial guess density matrix. If not given (the default), the kernel takes the density matrix generated by `mf.get_init_guess`.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

Returns: A list : scf_conv, e_tot, mo_energy, mo_coeff, mo_occ

scf_conv [bool] True means SCF converged

e_tot [float] Hartree-Fock energy of last iteration

mo_energy [1D float array] Orbital energies. Depending the eig function provided by mf object, the orbital energies may NOT be sorted.

mo_coeff [2D array] Orbital coefficients.

mo_occ [1D array] Orbital occupancies. The occupancies may NOT be sorted from large to small.

Examples:

```
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> conv, e, mo_e, mo, mo_occ = scf.hf.kernel(scf.hf.SCF(mol), dm0=numpy.eye(mol.
↪nao_nr()))
>>> print('conv = %s, E(HF) = %.12f' % (conv, e))
conv = True, E(HF) = -1.081170784378
```

pyscf.scf.hf.**level_shift**(*s*, *d*, *f*, *factor*)

Apply level shift Δ to virtual orbitals

$$FC = SCE \tag{1.1}$$

$$F = F + SC\Lambda C^{\dagger}S \tag{1.2}$$

$$\Lambda_{ij} = \begin{cases} \delta_{ij}\Delta & i \in \text{virtual} \\ 0 & \text{otherwise} \end{cases} \tag{1.3}$$

Returns: New Fock matrix, 2D ndarray

pyscf.scf.hf.**make_rdm1**(*mo_coeff*, *mo_occ*)

One-particle density matrix in AO representation

Args:

mo_coeff [2D ndarray] Orbital coefficients. Each column is one orbital.

mo_occ [1D ndarray] Occupancy

pyscf.scf.hf.**mulliken_meta**(*mol*, *dm*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)

Mulliken population analysis, based on meta-Lowdin AOs. In the meta-lowdin, the AOs are grouped in three sets: core, valence and Rydberg, the orthogonalization are carreid out within each subsets.

Args: mol : an instance of `Mole`

dm [ndarray or 2-item list of ndarray] Density matrix. ROHF dm is a 2-item list of 2D array

Kwargs: verbose : int or instance of `lib.logger.Logger`

pre_orth_method [str] Pre-orthogonalization, which localized GTOs for each atom. To obtain the occupied and unoccupied atomic shells, there are three methods

'ano' : Project GTOs to ANO basis
'minao' : Project GTOs to MINAO basis
'scf' : Fraction-averaged RHF

pyscf.scf.hf.**mulliken_pop**(*mol*, *dm*, *s=None*, *verbose=5*)

Mulliken population analysis

$$M_{ij} = D_{ij} S_{ji}$$

Mulliken charges

$$\delta_i = \sum_j M_{ij}$$

pyscf.scf.hf.**mulliken_pop_meta_lowdin_ao**(*mol*, *dm*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)

Mulliken population analysis, based on meta-Lowdin AOs. In the meta-lowdin, the AOs are grouped in three sets: core, valence and Rydberg, the orthogonalization are carreid out within each subsets.

Args: mol : an instance of `Mole`

dm [ndarray or 2-item list of ndarray] Density matrix. ROHF dm is a 2-item list of 2D array

Kwargs: verbose : int or instance of `lib.logger.Logger`

pre_orth_method [str] Pre-orthogonalization, which localized GTOs for each atom. To obtain the occupied and unoccupied atomic shells, there are three methods

'ano' : Project GTOs to ANO basis
'minao' : Project GTOs to MINAO basis
'scf' : Fraction-averaged RHF

---

class pyscf.scf.uhf.**UHF**(*mol*)

SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for UHF:

nelec [(int, int)] If given, freeze the number of (alpha,beta) electrons to the given value.

level_shift [number or two-element list] level shift (in Eh) for alpha and beta Fock if two-element list is given.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> print('S^2 = %.7f, 2S+1 = %.7f' % mf.spin_square())
S^2 = 0.7570150, 2S+1 = 2.0070027
```

canonicalize(*mf*, *mo_coeff*, *mo_occ*, *fock=None*)

Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separatedly (without change occupancy).

det_ovlp(*mo1*, *mo2*, *occ1*, *occ2*, *ovlp=None*)

Calculate the overlap between two different determinants. It is the product of single values of molecular orbital overlap matrix.

$$S_{12} = \langle \Psi_A | \Psi_B \rangle = (\det \mathbf{U})(\det \mathbf{V}^\dagger) \prod_{i=1}^{2N} \lambda_{ii}$$

where $\mathbf{U}, \mathbf{V}, \lambda$ are unitary matrices and single values generated by single value decomposition(SVD) of the overlap matrix $\mathbf{O}$ which is the overlap matrix of two sets of molecular orbitals:

$$\mathbf{U}^\dagger \mathbf{O} \mathbf{V} = \mathbf{\Lambda}$$

Args:

mo1, mo2 [2D ndarrays] Molecualr orbital coefficients

occ1, occ2: 2D ndarrays occupation numbers

Return:

A list: the product of single values: float x_a, x_b: 1D ndarrays $\mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{V}^\dagger$ They are used to calculate asymmetric density matrix

dip_moment (*mol=None*, *dm=None*, *unit_symbol=None*, *verbose=3*)
Dipole moment calculation

$$\mu_x = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A$$
$$\mu_y = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A$$
$$\mu_z = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A$$

where μ_x, μ_y, μ_z are the x, y and z components of dipole moment

Args: mol: an instance of `Mole`

dm [a list of 2D ndarrays] a list of density matrices

Return: A list: the dipole moment on x, y and z component

energy_elec (*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Electronic energy of Unrestricted Hartree-Fock

Returns: Hartree-Fock electronic energy and the 2-electron part contribution

get_jk (*mol=None*, *dm=None*, *hermi=1*)
Coulomb (J) and exchange (K)

Args:

dm [a list of 2D arrays or a list of 3D arrays] (alpha_dm, beta_dm) or (alpha_dms, beta_dms)

get_veff (*mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Unrestricted Hartree-Fock potential matrix of alpha and beta spins, for the given density matrix

$$V_{ij}^\alpha = \sum_{kl}(ij|kl)(\gamma_{lk}^\alpha + \gamma_{lk}^\beta) - \sum_{kl}(il|kj)\gamma_{lk}^\alpha$$
$$V_{ij}^\beta = \sum_{kl}(ij|kl)(\gamma_{lk}^\alpha + \gamma_{lk}^\beta) - \sum_{kl}(il|kj)\gamma_{lk}^\beta$$

Args: mol : an instance of `Mole`

dm [a list of ndarrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. When it is not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric

1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: $V_{hf} = (V^{\alpha}, V^{\beta})$. V^{α} (and V^{β}) can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dmsa = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dmsb = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dms = numpy.vstack((dmsa,dmsb))
>>> dms.shape
(6, 2, 2)
>>> vhfa, vhfb = scf.uhf.get_veff(mol, dms, hermi=0)
>>> vhfa.shape
(3, 2, 2)
>>> vhfb.shape
(3, 2, 2)
```

init_guess_by_minao(*mol=None*, *breaksym=True*)
Initial guess in terms of the overlap to minimal basis.

make_asym_dm(*mo1*, *mo2*, *occ1*, *occ2*, *x*)
One-particle asymmetric density matrix

Args:

mo1, mo2 [2D ndarrays] Molecualr orbital coefficients

occ1, occ2: 2D ndarrays Occupation numbers

x: 2D ndarrays $\mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{V}^{\dagger}$. See also *det_ovlp()*

Return: A list of 2D ndarrays for alpha and beta spin

Examples:

```
>>> mf1 = scf.UHF(gto.M(atom='H 0 0 0; F 0 0 1.3', basis='ccpvdz')).run()
>>> mf2 = scf.UHF(gto.M(atom='H 0 0 0; F 0 0 1.4', basis='ccpvdz')).run()
>>> s = gto.intor_cross('int1e_ovlp_sph', mf1.mol, mf2.mol)
>>> det, x = det_ovlp(mf1.mo_coeff, mf1.mo_occ, mf2.mo_coeff, mf2.mo_occ, s)
>>> adm = make_asym_dm(mf1.mo_coeff, mf1.mo_occ, mf2.mo_coeff, mf2.mo_occ, x)
>>> adm.shape
(2, 19, 19)
```

make_rdm1(*mo_coeff=None*, *mo_occ=None*)
One-particle density matrix

Returns: A list of 2D ndarrays for alpha and beta spins

spin_square(*mo_coeff=None*, *s=None*)
Spin of the given UHF orbitals

$$S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2$$

where $S_+ = \sum_i S_{i+}$ is effective for all beta occupied orbitals; $S_- = \sum_i S_{i-}$ is effective for all alpha occupied orbitals.

1.**There are two possibilities for** S_+S_-

(a) same electron $S_+S_- = \sum_i s_{i+}s_{i-}$,

$$\sum_i \langle UHF|s_{i+}s_{i-}|UHF\rangle = \sum_{pq}\langle p|s_+s_-|q\rangle\gamma_{qp} = n_\alpha$$

2) different electrons $S_+S_- = \sum s_{i+}s_{j-}, (i \neq j)$. There are in total $n(n-1)$ terms. As a two-particle operator,

$$\langle S_+S_-\rangle = \langle ij|s_+s_-|ij\rangle - \langle ij|s_+s_-|ji\rangle = -\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle$$

2.**Similarly, for** S_-S_+

(a) same electron

$$\sum_i \langle s_{i-}s_{i+}\rangle = n_\beta$$

(a) different electrons

$$\langle S_-S_+\rangle = -\langle i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta\rangle$$

2.**For** S_z^2

(a) same electron

$$\langle s_z^2\rangle = \frac{1}{4}(n_\alpha + n_\beta)$$

(a) different electrons

$$\begin{aligned}
&\frac{1}{2}\sum_{ij}(\langle ij|2s_{z1}s_{z2}|ij\rangle - \langle ij|2s_{z1}s_{z2}|ji\rangle) \\
&= \frac{1}{4}(\langle i^\alpha|i^\alpha\rangle\langle j^\alpha|j^\alpha\rangle - \langle i^\alpha|i^\alpha\rangle\langle j^\beta|j^\beta\rangle - \langle i^\beta|i^\beta\rangle\langle j^\alpha|j^\alpha\rangle + \langle i^\beta|i^\beta\rangle\langle j^\beta|j^\beta\rangle) \\
&- \frac{1}{4}(\langle i^\alpha|j^\alpha\rangle\langle j^\alpha|i^\alpha\rangle + \langle i^\beta|j^\beta\rangle\langle j^\beta|i^\beta\rangle) \\
&= \frac{1}{4}(n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2) - \frac{1}{4}(n_\alpha + n_\beta) \\
&= \frac{1}{4}((n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta))
\end{aligned}$$

In total

$$\langle S^2\rangle = \frac{1}{2}(n_\alpha - \sum_{ij}\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle + n_\beta - \sum_{ij}\langle i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta\rangle) + \frac{1}{4}(n_\alpha - n_\beta)^2$$

Args:

mo [a list of 2 ndarrays] Occupied alpha and occupied beta orbitals

Kwargs:

s [ndarray] AO overlap

Returns: A list of two floats. The first is the expectation value of S^2. The second is the corresponding 2S+1

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> mo = (mf.mo_coeff[0][:,mf.mo_occ[0]>0], mf.mo_coeff[1][:,mf.mo_occ[1]>0])
>>> print('S^2 = %.7f, 2S+1 = %.7f' % spin_square(mo, mol.intor('int1e_ovlp_
↪sph')))
S^2 = 0.7570150, 2S+1 = 2.0070027
```

pyscf.scf.uhf.**analyze**(*mf*, *verbose=5*, ***kwargs*)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Dipole moment

pyscf.scf.uhf.**canonicalize**(*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separatedly (without change occupancy).

pyscf.scf.uhf.**det_ovlp**(*mo1*, *mo2*, *occ1*, *occ2*, *ovlp*)
Calculate the overlap between two different determinants. It is the product of single values of molecular orbital overlap matrix.

$$S_{12} = \langle \Psi_A | \Psi_B \rangle = (\det \mathbf{U})(\det \mathbf{V}^\dagger) \prod_{i=1}^{2N} \lambda_{ii}$$

where $\mathbf{U}, \mathbf{V}, \lambda$ are unitary matrices and single values generated by single value decomposition(SVD) of the overlap matrix $\mathbf{O}$ which is the overlap matrix of two sets of molecular orbitals:

$$\mathbf{U}^\dagger \mathbf{O} \mathbf{V} = \mathbf{\Lambda}$$

Args:

mo1, mo2 [2D ndarrays] Molecualr orbital coefficients

occ1, occ2: 2D ndarrays occupation numbers

Return:

A list: the product of single values: float x_a, x_b: 1D ndarrays $\mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{V}^\dagger$ They are used to calculate asymmetric density matrix

pyscf.scf.uhf.**dip_moment**(*mol*, *dm*, *unit_symbol='Debye'*, *verbose=3*)
Dipole moment calculation

$$\mu_x = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|x|\mu) + \sum_A Q_A X_A$$
$$\mu_y = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|y|\mu) + \sum_A Q_A Y_A$$
$$\mu_z = -\sum_\mu \sum_\nu P_{\mu\nu}(\nu|z|\mu) + \sum_A Q_A Z_A$$

where μ_x, μ_y, μ_z are the x, y and z components of dipole moment

Args: mol: an instance of `Mole`

dm [a list of 2D ndarrays] a list of density matrices

Return: A list: the dipole moment on x, y and z component

pyscf.scf.uhf.**energy_elec**(*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Electronic energy of Unrestricted Hartree-Fock

Returns: Hartree-Fock electronic energy and the 2-electron part contribution

pyscf.scf.uhf.**get_grad**(*mo_coeff*, *mo_occ*, *fock_ao*)
UHF Gradients

pyscf.scf.uhf.**get_veff**(*mol*, *dm*, *dm_last=0*, *vhf_last=0*, *hermi=1*, *vhfopt=None*)
Unrestricted Hartree-Fock potential matrix of alpha and beta spins, for the given density matrix

$$V_{ij}^{\alpha} = \sum_{kl}(ij|kl)(\gamma_{lk}^{\alpha} + \gamma_{lk}^{\beta}) - \sum_{kl}(il|kj)\gamma_{lk}^{\alpha}$$
$$V_{ij}^{\beta} = \sum_{kl}(ij|kl)(\gamma_{lk}^{\alpha} + \gamma_{lk}^{\beta}) - \sum_{kl}(il|kj)\gamma_{lk}^{\beta}$$

Args: mol : an instance of `Mole`

dm [a list of ndarrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. When it is not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference HF potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

Returns: $V_{hf} = (V^{\alpha}, V^{\beta})$. V^{α} (and V^{β}) can be a list matrices, corresponding to the input density matrices.

Examples:

```
>>> import numpy
>>> from pyscf import gto, scf
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> dmsa = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dmsb = numpy.random.random((3,mol.nao_nr(),mol.nao_nr()))
>>> dms = numpy.vstack((dmsa,dmsb))
>>> dms.shape
(6, 2, 2)
>>> vhfa, vhfb = scf.uhf.get_veff(mol, dms, hermi=0)
>>> vhfa.shape
(3, 2, 2)
>>> vhfb.shape
(3, 2, 2)
```

pyscf.scf.uhf.**init_guess_by_minao**(*mol*, *breaksym=True*)
Generate initial guess density matrix based on ANO basis, then project the density matrix to the basis set defined by `mol`

Returns: Density matrices, a list of 2D ndarrays for alpha and beta spins

pyscf.scf.uhf.**make_asym_dm**(*mo1*, *mo2*, *occ1*, *occ2*, *x*)
One-particle asymmetric density matrix

Args:

mo1, mo2 [2D ndarrays] Molecualr orbital coefficients

occ1, occ2: 2D ndarrays Occupation numbers

x: 2D ndarrays $\mathbf{U}\mathbf{\Lambda}^{-1}\mathbf{V}^{\dagger}$. See also *det_ovlp()*

Return: A list of 2D ndarrays for alpha and beta spin

Examples:

```
>>> mf1 = scf.UHF(gto.M(atom='H 0 0 0; F 0 0 1.3', basis='ccpvdz')).run()
>>> mf2 = scf.UHF(gto.M(atom='H 0 0 0; F 0 0 1.4', basis='ccpvdz')).run()
>>> s = gto.intor_cross('int1e_ovlp_sph', mf1.mol, mf2.mol)
>>> det, x = det_ovlp(mf1.mo_coeff, mf1.mo_occ, mf2.mo_coeff, mf2.mo_occ, s)
>>> adm = make_asym_dm(mf1.mo_coeff, mf1.mo_occ, mf2.mo_coeff, mf2.mo_occ, x)
>>> adm.shape
(2, 19, 19)
```

pyscf.scf.uhf.**make_rdm1**(*mo_coeff*, *mo_occ*)
One-particle density matrix

Returns: A list of 2D ndarrays for alpha and beta spins

pyscf.scf.uhf.**mulliken_meta**(*mol*, *dm_ao*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)
Mulliken population analysis, based on meta-Lowdin AOs.

pyscf.scf.uhf.**mulliken_pop**(*mol*, *dm*, *s=None*, *verbose=5*)
Mulliken population analysis

pyscf.scf.uhf.**mulliken_pop_meta_lowdin_ao**(*mol*, *dm_ao*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)
Mulliken population analysis, based on meta-Lowdin AOs.

pyscf.scf.uhf.**rhf_to_uhf**(*mf*)
Create UHF object based on the RHF object

pyscf.scf.uhf.**spin_square**(*mo*, *s=1*)
Spin of the given UHF orbitals

$$S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2$$

where $S_+ = \sum_i S_{i+}$ is effective for all beta occupied orbitals; $S_- = \sum_i S_{i-}$ is effective for all alpha occupied orbitals.

1.**There are two possibilities for** S_+S_-

(a) same electron $S_+S_- = \sum_i s_{i+}s_{i-}$,

$$\sum_i \langle UHF|s_{i+}s_{i-}|UHF\rangle = \sum_{pq}\langle p|s_+s_-|q\rangle\gamma_{qp} = n_\alpha$$

2) different electrons $S_+S_- = \sum s_{i+}s_{j-}, (i \neq j)$. There are in total $n(n-1)$ terms. As a two-particle operator,

$$\langle S_+S_-\rangle = \langle ij|s_+s_-|ij\rangle - \langle ij|s_+s_-|ji\rangle = -\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle$$

2.**Similarly, for** S_-S_+

(a) same electron

$$\sum_i \langle s_{i-} s_{i+} \rangle = n_\beta$$

(a) different electrons

$$\langle S_- S_+ \rangle = -\langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle$$

2.**For** S_z^2

(a) same electron

$$\langle s_z^2 \rangle = \frac{1}{4}(n_\alpha + n_\beta)$$

(a) different electrons

$$\begin{aligned}
&\frac{1}{2}\sum_{ij}(\langle ij|2s_{z1}s_{z2}|ij\rangle - \langle ij|2s_{z1}s_{z2}|ji\rangle) \\
&= \frac{1}{4}(\langle i^\alpha|i^\alpha\rangle\langle j^\alpha|j^\alpha\rangle - \langle i^\alpha|i^\alpha\rangle\langle j^\beta|j^\beta\rangle - \langle i^\beta|i^\beta\rangle\langle j^\alpha|j^\alpha\rangle + \langle i^\beta|i^\beta\rangle\langle j^\beta|j^\beta\rangle) \\
&- \frac{1}{4}(\langle i^\alpha|j^\alpha\rangle\langle j^\alpha|i^\alpha\rangle + \langle i^\beta|j^\beta\rangle\langle j^\beta|i^\beta\rangle) \\
&= \frac{1}{4}(n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2) - \frac{1}{4}(n_\alpha + n_\beta) \\
&= \frac{1}{4}((n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta))
\end{aligned}$$

In total

$$\langle S^2 \rangle = \frac{1}{2}(n_\alpha - \sum_{ij}\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle + n_\beta - \sum_{ij}\langle i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta\rangle) + \frac{1}{4}(n_\alpha - n_\beta)^2$$

Args:

mo [a list of 2 ndarrays] Occupied alpha and occupied beta orbitals

Kwargs:

s [ndarray] AO overlap

Returns: A list of two floats. The first is the expectation value of S^2. The second is the corresponding 2S+1

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> mo = (mf.mo_coeff[0][:,mf.mo_occ[0]>0], mf.mo_coeff[1][:,mf.mo_occ[1]>0])
>>> print('S^2 = %.7f, 2S+1 = %.7f' % spin_square(mo, mol.intor('int1e_ovlp_sph
↪')))
S^2 = 0.7570150, 2S+1 = 2.0070027
```

Non-relativistic restricted Hartree-Fock with point group symmetry.

The symmetry are not handled in a separate data structure. Note that during the SCF iteration, the orbitals are grouped in terms of symmetry irreps. But the orbitals in the result are sorted based on the orbital energies. Function symm.label_orb_symm can be used to detect the symmetry of the molecular orbitals.

class pyscf.scf.hf_symm.**RHF**(*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to Mole.verbose

max_memory [float or int] Allowed memory in MB. Default equals to Mole.max_memory

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in scf.diis] Default is diis.SCF_DIIS. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function locals(), so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for symmetry allowed RHF:

irrep_nelec [dict] Specify the number of electrons for particular irrep {'ir_name':int,...}. For the irreps not listed in this dict, the program will choose the occupancy based on the orbital energies.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', symmetry=True,␣
↪verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
-76.016789472074251
>>> mf.get_irrep_nelec()
{'A1': 6, 'A2': 0, 'B1': 2, 'B2': 2}
>>> mf.irrep_nelec = {'A2': 2}
>>> mf.scf()
-72.768201804695622
>>> mf.get_irrep_nelec()
{'A1': 6, 'A2': 2, 'B1': 2, 'B2': 0}
```

canonicalize (*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix in occupied, open, virtual subspaces separatedly (without change occupancy).

eig (*mf*, *h*, *s*)
Solve generalized eigenvalue problem, for each irrep. The eigenvalues and eigenvectors are not sorted to ascending order. Instead, they are grouped based on irreps.

get_irrep_nelec (*mol=None*, *mo_coeff=None*, *mo_occ=None*, *s=None*)
Electron numbers for each irreducible representation.

Args:

mol [an instance of `Mole`] To provide irrep_id, and spin-adapted basis

mo_coeff [2D ndarray] Regular orbital coefficients, without grouping for irreps

mo_occ [1D ndarray] Regular occupancy, without grouping for irreps

Returns:

irrep_nelec [dict] The number of electrons for each irrep {'ir_name':int,...}.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz',␣
↪symmetry=True, verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
-76.016789472074251
>>> scf.hf_symm.get_irrep_nelec(mol, mf.mo_coeff, mf.mo_occ)
{'A1': 6, 'A2': 0, 'B1': 2, 'B2': 2}
```

get_occ (*mo_energy=None*, *mo_coeff=None*)
We assumed mo_energy are grouped by symmetry irreps, (see function self.eig). The orbitals are sorted after SCF.

class `pyscf.scf.hf_symm.`**ROHF** (*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for symmetry allowed ROHF:

irrep_nelec [dict] Specify the number of alpha/beta electrons for particular irrep {'ir_name':(int,int), ...}. For the irreps not listed in these dicts, the program will choose the occupancy based on the orbital energies.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', symmetry=True,␣
↪charge=1, spin=1, verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
-75.619358861084052
>>> mf.get_irrep_nelec()
{'A1': (3, 3), 'A2': (0, 0), 'B1': (1, 1), 'B2': (1, 0)}
```

```
>>> mf.irrep_nelec = {'B1': (1, 0)}
>>> mf.scf()
-75.425669486776457
>>> mf.get_irrep_nelec()
{'A1': (3, 3), 'A2': (0, 0), 'B1': (1, 0), 'B2': (1, 1)}
```

canonicalize (*mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix in occupied, open, virtual subspaces separatedly (without change occupancy).

eig (*mf*, *h*, *s*)
Solve generalized eigenvalue problem, for each irrep. The eigenvalues and eigenvectors are not sorted to ascending order. Instead, they are grouped based on irreps.

pyscf.scf.hf_symm.**analyze** (*mf*, *verbose=5*, ***kwargs*)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Occupancy for each irreps; Mulliken population analysis

pyscf.scf.hf_symm.**canonicalize** (*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the Fock matrix in occupied, open, virtual subspaces separatedly (without change occupancy).

pyscf.scf.hf_symm.**eig** (*mf*, *h*, *s*)
Solve generalized eigenvalue problem, for each irrep. The eigenvalues and eigenvectors are not sorted to ascending order. Instead, they are grouped based on irreps.

pyscf.scf.hf_symm.**get_irrep_nelec** (*mol*, *mo_coeff*, *mo_occ*, *s=None*)
Electron numbers for each irreducible representation.

Args:

mol [an instance of Mole] To provide irrep_id, and spin-adapted basis

mo_coeff [2D ndarray] Regular orbital coefficients, without grouping for irreps

mo_occ [1D ndarray] Regular occupancy, without grouping for irreps

Returns:

irrep_nelec [dict] The number of electrons for each irrep {'ir_name':int,...}.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', symmetry=True,␣
↪verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
-76.016789472074251
>>> scf.hf_symm.get_irrep_nelec(mol, mf.mo_coeff, mf.mo_occ)
{'A1': 6, 'A2': 0, 'B1': 2, 'B2': 2}
```

pyscf.scf.hf_symm.**so2ao_mo_coeff** (*so*, *irrep_mo_coeff*)
Transfer the basis of MO coefficients, from spin-adapted basis to AO basis

---

Non-relativistic unrestricted Hartree-Fock with point group symmetry.

class pyscf.scf.uhf_symm.**UHF** (*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default equals to `Mole.max_memory`

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for UHF:

nelec [(int, int)] If given, freeze the number of (alpha,beta) electrons to the given value.

level_shift [number or two-element list] level shift (in Eh) for alpha and beta Fock if two-element list is given.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> print('S^2 = %.7f, 2S+1 = %.7f' % mf.spin_square())
S^2 = 0.7570150, 2S+1 = 2.0070027
```

Attributes for symmetry allowed UHF:

irrep_nelec [dict] Specify the number of alpha/beta electrons for particular irrep {'ir_name':(int,int), ...}. For the irreps not listed in these dicts, the program will choose the occupancy based on the orbital energies.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', symmetry=True,␣
↪charge=1, spin=1, verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
-75.623975516256692
>>> mf.get_irrep_nelec()
{'A1': (3, 3), 'A2': (0, 0), 'B1': (1, 1), 'B2': (1, 0)}
>>> mf.irrep_nelec = {'B1': (1, 0)}
>>> mf.scf()
-75.429189192031131
>>> mf.get_irrep_nelec()
{'A1': (3, 3), 'A2': (0, 0), 'B1': (1, 0), 'B2': (1, 1)}
```

canonicalize(*mf*, *mo_coeff*, *mo_occ*, *fock=None*)

Canonicalization diagonalizes the UHF Fock matrix in occupied, virtual subspaces separatedly (without change occupancy).

get_irrep_nelec(*mol=None*, *mo_coeff=None*, *mo_occ=None*, *s=None*)

Alpha/beta electron numbers for each irreducible representation.

Args:

mol [an instance of `Mole`] To provide irrep_id, and spin-adapted basis

mo_occ [a list of 1D ndarray] Regular occupancy, without grouping for irreps

mo_coeff [a list of 2D ndarray] Regular orbital coefficients, without grouping for irreps

Returns:

irrep_nelec [dict] The number of alpha/beta electrons for each irrep {'ir_name':(int,int), ...}.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz',␣
↪symmetry=True, charge=1, spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.scf()
-75.623975516256721
>>> scf.uhf_symm.get_irrep_nelec(mol, mf.mo_coeff, mf.mo_occ)
{'A1': (3, 3), 'A2': (0, 0), 'B1': (1, 1), 'B2': (1, 0)}
```

get_occ(*mo_energy=None*, *mo_coeff=None*)
We assumed mo_energy are grouped by symmetry irreps, (see function self.eig). The orbitals are sorted after SCF.

pyscf.scf.uhf_symm.**canonicalize**(*mf*, *mo_coeff*, *mo_occ*, *fock=None*)
Canonicalization diagonalizes the UHF Fock matrix in occupied, virtual subspaces separatedly (without change occupancy).

pyscf.scf.uhf_symm.**get_irrep_nelec**(*mol*, *mo_coeff*, *mo_occ*, *s=None*)
Alpha/beta electron numbers for each irreducible representation.

Args:

mol [an instance of Mole] To provide irrep_id, and spin-adapted basis

mo_occ [a list of 1D ndarray] Regular occupancy, without grouping for irreps

mo_coeff [a list of 2D ndarray] Regular orbital coefficients, without grouping for irreps

Returns:

irrep_nelec [dict] The number of alpha/beta electrons for each irrep {'ir_name':(int,int), ...}.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', symmetry=True,␣
↪charge=1, spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.scf()
-75.623975516256721
>>> scf.uhf_symm.get_irrep_nelec(mol, mf.mo_coeff, mf.mo_occ)
{'A1': (3, 3), 'A2': (0, 0), 'B1': (1, 1), 'B2': (1, 0)}
```

Relativistic Hartree-Fock

Dirac Hartree-Fock

pyscf.scf.dhf.**DHF**
alias of *UHF*

class pyscf.scf.dhf.**RHF**(*mol*)
Dirac-RHF

make_rdm1(*mo_coeff=None*, *mo_occ=None*)
D/2 = psi_i^dagpsi_i = psi_{Ti}^dagpsi_{Ti} D(UHF) = psi_i^dagpsi_i + psi_{Ti}^dagpsi_{Ti} RHF average the density of spin up and spin down: D(RHF) = (D(UHF) + T[D(UHF)])/2

class pyscf.scf.dhf.**UHF**(*mol*)
SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to Mole.verbose

max_memory [float or int] Allowed memory in MB. Default equals to Mole.max_memory

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in `scf.diis`] Default is `diis.SCF_DIIS`. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for Dirac-Hartree-Fock

with_ssss [bool, for Dirac-Hartree-Fock only] If False, ignore small component integrals (SS|SS). Default is True.

with_gaunt [bool, for Dirac-Hartree-Fock only] Default is False.

with_breit [bool, for Dirac-Hartree-Fock only] Gaunt + gauge term. Default is False.

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> e0 = mf.scf()
>>> mf = scf.DHF(mol)
>>> e1 = mf.scf()
>>> print('Relativistic effects = %.12f' % (e1-e0))
Relativistic effects = -0.000008854205
```

get_veff(*mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Dirac-Coulomb

init_guess_by_minao(*mol=None*)
Initial guess in terms of the overlap to minimal basis.

pyscf.scf.dhf.**get_grad**(*mo_coeff*, *mo_occ*, *fock_ao*)
DHF Gradients

pyscf.scf.dhf.**init_guess_by_1e**(*mol*)
Initial guess from one electron system.

pyscf.scf.dhf.**init_guess_by_atom**(*mol*)
Initial guess from atom calculation.

pyscf.scf.dhf.**init_guess_by_minao**(*mol*)
Initial guess in terms of the overlap to minimal basis.

pyscf.scf.dhf.**kernel**(*mf*, *conv_tol=1e-09*, *conv_tol_grad=None*, *dump_chk=True*, *dm0=None*, *callback=None*, *conv_check=True*)
the modified SCF kernel for Dirac-Hartree-Fock. In this kernel, the SCF is carried out in three steps. First the 2-electron part is approximated by large component integrals (LL|LL); Next, (SS|LL) the interaction between large and small components are added; Finally, converge the SCF with the small component contributions (SS|SS)

pyscf.scf.dhf.**time_reversal_matrix**(*mol*, *mat*)
T(A_ij) = A[T(i),T(j)]^*

addons

pyscf.scf.addons.**convert_to_rhf**(*mf*, *out=None*, *convert_df=None*)
Convert the given mean-field object to the corresponding restricted HF/KS object

Args: mf : SCF object

Kwargs

convert_df [bool] Whether to convert the DF-SCF object to the normal SCF object. This conversion is not applied by default.

Returns: An unrestricted SCF object

pyscf.scf.addons.**convert_to_uhf**(*mf*, *out=None*, *convert_df=None*)
Convert the given mean-field object to the corresponding unrestricted HF/KS object

Args: mf : SCF object

Kwargs

convert_df [bool] Whether to convert the DF-SCF object to the normal SCF object. This conversion is not applied by default.

Returns: An unrestricted SCF object

pyscf.scf.addons.**dynamic_level_shift**(*mf*, *factor=1.0*)
Dynamically change the level shift in each SCF cycle. The level shift value is set to (HF energy change * factor)

pyscf.scf.addons.**dynamic_level_shift_**(*mf*, *factor=1.0*)
Dynamically change the level shift in each SCF cycle. The level shift value is set to (HF energy change * factor)

pyscf.scf.addons.**dynamic_sz_**(*mf*)
For UHF, allowing the Sz value being changed during SCF iteration. Determine occupation of alpha and beta electrons based on energy spectrum

pyscf.scf.addons.**float_occ**(*mf*)
For UHF, allowing the Sz value being changed during SCF iteration. Determine occupation of alpha and beta electrons based on energy spectrum

pyscf.scf.addons.**float_occ_**(*mf*)
For UHF, allowing the Sz value being changed during SCF iteration. Determine occupation of alpha and beta electrons based on energy spectrum

pyscf.scf.addons.**mom_occ**(*mf*, *occorb*, *setocc*)
Use maximum overlap method to determine occupation number for each orbital in every iteration. It can be applied to unrestricted HF/KS and restricted open-shell HF/KS.

pyscf.scf.addons.**mom_occ_**(*mf*, *occorb*, *setocc*)
Use maximum overlap method to determine occupation number for each orbital in every iteration. It can be applied to unrestricted HF/KS and restricted open-shell HF/KS.

pyscf.scf.addons.**project_mo_nr2nr**(*mol1*, *mo1*, *mol2*)
Project orbital coefficients

$$|\psi 1>=|AO1>C1$$
$$|\psi 2>=P|\psi 1>=|AO2>S^{-1}<AO2|AO1>C1=|AO2>C2$$
$$C2=S^{-1}<AO2|AO1>C1$$

pyscf.scf.addons.**symm_allow_occ**(*mf*, *tol=0.001*)
search the unoccupied orbitals, choose the lowest sets which do not break symmetry as the occupied orbitals

pyscf.scf.addons.**symm_allow_occ_**(*mf*, *tol=0.001*)
search the unoccupied orbitals, choose the lowest sets which do not break symmetry as the occupied orbitals

---

pyscf.scf.chkfile.**dump_scf**(*mol*, *chkfile*, *e_tot*, *mo_energy*, *mo_coeff*, *mo_occ*, *overwrite_mol=True*)
save temporary results

DIIS

class pyscf.scf.diis.**ADIIS**(*dev=None*, *filename=None*)
Ref: JCP, 132, 054109

class pyscf.scf.diis.**EDIIS**(*dev=None*, *filename=None*)
SCF-EDIIS Ref: JCP 116, 8255

1.7 ao2mo — Integral transformations

1.7.1 General Integral transformation module

Simple usage:

```
>>> from pyscf import gto, scf, ao2mo
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> mf = scf.RHF(mol).run()
>>> mo_ints = ao2mo.kernel(mol, mf.mo_coeff)
```

1.7.2 incore

pyscf.ao2mo.incore.**full**(*eri_ao*, *mo_coeff*, *verbose=0*, *compact=True*, ***kwargs*)
MO integral transformation for the given orbital.

Args:

eri_ao [ndarray] AO integrals, can be either 8-fold or 4-fold symmetry.

mo_coeff [ndarray] Transform (ij|kl) with the same set of orbitals.

Kwargs:

verbose [int] Print level

compact [bool] When compact is True, the returned MO integrals have 4-fold symmetry. Otherwise, return the "plain" MO integrals.

Returns: 2D array of transformed MO integrals. The MO integrals may or may not have the permutation symmetry (controlled by the kwargs compact)

Examples:

```
>>> from pyscf import gto
>>> from pyscf.scf import _vhf
>>> from pyscf import ao2mo
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> eri = mol.intor('int2e_sph', aosym='s8')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> eri1 = ao2mo.incore.full(eri, mo1)
>>> print(eri1.shape)
(55, 55)
>>> eri1 = ao2mo.incore.full(eri, mo1, compact=False)
>>> print(eri1.shape)
(100, 100)
```

pyscf.ao2mo.incore.**general**(*eri_ao*, *mo_coeffs*, *verbose=0*, *compact=True*, ***kwargs*)
For the given four sets of orbitals, transfer the 8-fold or 4-fold 2e AO integrals to MO integrals.

Args:

eri_ao [ndarray] AO integrals, can be either 8-fold or 4-fold symmetry.

mo_coeffs [4-item list of ndarray] Four sets of orbital coefficients, corresponding to the four indices of (ij|kl)

Kwargs:

verbose [int] Print level

compact [bool] When compact is True, depending on the four oribital sets, the returned MO integrals has (up to 4-fold) permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

Returns: 2D array of transformed MO integrals. The MO integrals may or may not have the permutation symmetry, depending on the given orbitals, and the kwargs compact. If the four sets of orbitals are identical, the MO integrals will at most have 4-fold symmetry.

Examples:

```
>>> from pyscf import gto
>>> from pyscf.scf import _vhf
>>> from pyscf import ao2mo
```

```
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> eri = mol.intor('int2e_sph', aosym='s8')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> mo2 = numpy.random.random((mol.nao_nr(), 8))
>>> mo3 = numpy.random.random((mol.nao_nr(), 6))
>>> mo4 = numpy.random.random((mol.nao_nr(), 4))
>>> eri1 = ao2mo.incore.general(eri, (mo1,mo2,mo3,mo4))
>>> print(eri1.shape)
(80, 24)
>>> eri1 = ao2mo.incore.general(eri, (mo1,mo2,mo3,mo3))
>>> print(eri1.shape)
(80, 21)
>>> eri1 = ao2mo.incore.general(eri, (mo1,mo2,mo3,mo3), compact=False)
>>> print(eri1.shape)
(80, 36)
>>> eri1 = ao2mo.incore.general(eri, (mo1,mo1,mo2,mo2))
>>> print(eri1.shape)
(55, 36)
>>> eri1 = ao2mo.incore.general(eri, (mo1,mo2,mo1,mo2))
>>> print(eri1.shape)
(80, 80)
```

pyscf.ao2mo.incore.**half_e1**(*eri_ao*, *mo_coeffs*, *compact=True*)

Given two set of orbitals, half transform the (ij| pair of 8-fold or 4-fold AO integrals (ij|kl)

Args:

eri_ao [ndarray] AO integrals, can be either 8-fold or 4-fold symmetry.

mo_coeffs [list of ndarray] Two sets of orbital coefficients, corresponding to the i, j indices of (ij|kl)

Kwargs:

compact [bool] When compact is True, the returned MO integrals uses the highest possible permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

Returns: ndarray of transformed MO integrals. The MO integrals may or may not have the permutation symmetry, depending on the given orbitals, and the kwargs compact.

Examples:

```
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> eri = mol.intor('int2e_sph', aosym='s8')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> mo2 = numpy.random.random((mol.nao_nr(), 8))
>>> eri1 = ao2mo.incore.half_e1(eri, (mo1,mo2))
>>> eri1 = ao2mo.incore.half_e1(eri, (mo1,mo2))
>>> print(eri1.shape)
(80, 28)
>>> eri1 = ao2mo.incore.half_e1(eri, (mo1,mo2), compact=False)
>>> print(eri1.shape)
(80, 28)
>>> eri1 = ao2mo.incore.half_e1(eri, (mo1,mo1))
>>> print(eri1.shape)
(55, 28)
```

1.7.3 outcore

pyscf.ao2mo.outcore.**full**(*mol*, *mo_coeff*, *erifile*, *dataname='eri_mo'*, *tmpdir=None*, *intor='int2e_sph'*, *aosym='s4'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *verbose=2*, *compact=True*)

Transfer arbitrary spherical AO integrals to MO integrals for given orbitals

Args:

mol [`Mole` object] AO integrals will be generated in terms of mol._atm, mol._bas, mol._env

mo_coeff [ndarray] Transform (ij|kl) with the same set of orbitals.

erifile [str or h5py File or h5py Group object] To store the transformed integrals, in HDF5 format.

Kwargs:

dataname [str] The dataset name in the erifile (ref the hierarchy of HDF5 format http://www.hdfgroup.org/HDF5/doc1.6/UG/09_Groups.html). By assigning different dataname, the existed integral file can be reused. If the erifile contains the dataname, the new integrals data will overwrite the old one.

tmpdir [str] The directory where to temporarily store the intermediate data (the half-transformed integrals). By default, it's controlled by shell environment variable `TMPDIR`. The disk space requirement is about comp*mo_coeffs[0].shape[1]*mo_coeffs[1].shape[1]*nao**2

intor [str] Name of the 2-electron integral. Ref to `getints_by_shell()` for the complete list of available 2-electron integral names

aosym [int or str] Permutation symmetry for the AO integrals

4 or '4' or 's4': 4-fold symmetry (default)
'2ij' or 's2ij' : symmetry between i, j in (ij|kl)
'2kl' or 's2kl' : symmetry between k, l in (ij|kl)
1 or '1' or 's1': no symmetry
'a4ij' : 4-fold symmetry with anti-symmetry between i, j in (ij|kl) (TODO)
'a4kl' : 4-fold symmetry with anti-symmetry between k, l in (ij|kl) (TODO)
'a2ij' : anti-symmetry between i, j in (ij|kl) (TODO)
'a2kl' : anti-symmetry between k, l in (ij|kl) (TODO)

comp [int] Components of the integrals, e.g. int2e_ip_sph has 3 components.

max_memory [float or int] The maximum size of cache to use (in MB), large cache may **not** improve performance.

ioblk_size [float or int] The block size for IO, large block size may **not** improve performance

verbose [int] Print level

compact [bool] When compact is True, depending on the four oribital sets, the returned MO integrals has (up to 4-fold) permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

Returns: None

Examples:

```
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> import h5py
```

```
>>> def view(h5file, dataname='eri_mo'):
...     f5 = h5py.File(h5file)
...     print('dataset %s, shape %s' % (str(f5.keys()), str(f5[dataname].shape)))
...     f5.close()
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> ao2mo.outcore.full(mol, mo1, 'full.h5')
>>> view('full.h5')
dataset ['eri_mo'], shape (55, 55)
>>> ao2mo.outcore.full(mol, mo1, 'full.h5', dataname='new', compact=False)
>>> view('full.h5', 'new')
dataset ['eri_mo', 'new'], shape (100, 100)
>>> ao2mo.outcore.full(mol, mo1, 'full.h5', intor='int2e_ip1_sph', aosym='s1',␣
↪comp=3)
>>> view('full.h5')
dataset ['eri_mo', 'new'], shape (3, 100, 100)
>>> ao2mo.outcore.full(mol, mo1, 'full.h5', intor='int2e_ip1_sph', aosym='s2kl',␣
↪comp=3)
>>> view('full.h5')
dataset ['eri_mo', 'new'], shape (3, 100, 55)
```

pyscf.ao2mo.outcore.**full_iofree**(*mol*, *mo_coeff*, *intor='int2e_sph'*, *aosym='s4'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *verbose=2*, *compact=True*)

Transfer arbitrary spherical AO integrals to MO integrals for given orbitals This function is a wrap for `ao2mo.outcore.general()`. It's not really IO free. The returned MO integrals are held in memory. For backward compatibility, it is used to replace the non-existed function direct.full_iofree.

Args:

mol [`Mole` object] AO integrals will be generated in terms of mol._atm, mol._bas, mol._env

mo_coeff [ndarray] Transform (ij|kl) with the same set of orbitals.

erifile [str] To store the transformed integrals, in HDF5 format.

Kwargs

dataname [str] The dataset name in the erifile (ref the hierarchy of HDF5 format http://www.hdfgroup.org/HDF5/doc1.6/UG/09_Groups.html). By assigning different dataname, the existed integral file can be reused. If the erifile contains the dataname, the new integrals data will overwrite the old one.

tmpdir [str] The directory where to temporarily store the intermediate data (the half-transformed integrals). By default, it's controlled by shell environment variable `TMPDIR`. The disk space requirement is about comp*mo_coeffs[0].shape[1]*mo_coeffs[1].shape[1]*nao**2

intor [str] Name of the 2-electron integral. Ref to `getints_by_shell()` for the complete list of available 2-electron integral names

aosym [int or str] Permutation symmetry for the AO integrals

4 or '4' or 's4': 4-fold symmetry (default)
'2ij' or 's2ij' : symmetry between i, j in (ij|kl)
'2kl' or 's2kl' : symmetry between k, l in (ij|kl)
1 or '1' or 's1': no symmetry
'a4ij' : 4-fold symmetry with anti-symmetry between i, j in (ij|kl) (TODO)
'a4kl' : 4-fold symmetry with anti-symmetry between k, l in (ij|kl) (TODO)
'a2ij' : anti-symmetry between i, j in (ij|kl) (TODO)

'a2kl' : anti-symmetry between k, l in (ij|kl) (TODO)

comp [int] Components of the integrals, e.g. int2e_ip_sph has 3 components.

verbose [int] Print level

max_memory [float or int] The maximum size of cache to use (in MB), large cache may **not** improve performance.

ioblk_size [float or int] The block size for IO, large block size may **not** improve performance

verbose [int] Print level

compact [bool] When compact is True, depending on the four oribital sets, the returned MO integrals has (up to 4-fold) permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

Returns: 2D/3D MO-integral array. They may or may not have the permutation symmetry, depending on the given orbitals, and the kwargs compact. If the four sets of orbitals are identical, the MO integrals will at most have 4-fold symmetry.

Examples:

```
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> eri1 = ao2mo.outcore.full_iofree(mol, mo1)
>>> print(eri1.shape)
(55, 55)
>>> eri1 = ao2mo.outcore.full_iofree(mol, mo1, compact=False)
>>> print(eri1.shape)
(100, 100)
>>> eri1 = ao2mo.outcore.full_iofree(mol, mo1, intor='int2e_ip1_sph', aosym='s1',␣
↪comp=3)
>>> print(eri1.shape)
(3, 100, 100)
>>> eri1 = ao2mo.outcore.full_iofree(mol, mo1, intor='int2e_ip1_sph', aosym='s2kl
↪', comp=3)
>>> print(eri1.shape)
(3, 100, 55)
```

pyscf.ao2mo.outcore.**general**(*mol*, *mo_coeffs*, *erifile*, *dataname='eri_mo'*, *tmpdir=None*, *intor='int2e_sph'*, *aosym='s4'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *verbose=2*, *compact=True*)

For the given four sets of orbitals, transfer arbitrary spherical AO integrals to MO integrals on the fly.

Args:

mol [Mole object] AO integrals will be generated in terms of mol._atm, mol._bas, mol._env

mo_coeffs [4-item list of ndarray] Four sets of orbital coefficients, corresponding to the four indices of (ij|kl)

erifile [str or h5py File or h5py Group object] To store the transformed integrals, in HDF5 format.

Kwargs

dataname [str] The dataset name in the erifile (ref the hierarchy of HDF5 format http://www.hdfgroup.org/HDF5/doc1.6/UG/09_Groups.html). By assigning different dataname, the existed integral file can be reused. If the erifile contains the dataname, the new integrals data will overwrite the old one.

tmpdir [str] The directory where to temporarily store the intermediate data (the half-transformed integrals). By default, it's controlled by shell environment variable `TMPDIR`. The disk space requirement is about comp*mo_coeffs[0].shape[1]*mo_coeffs[1].shape[1]*nao**2

intor [str] Name of the 2-electron integral. Ref to `getints_by_shell()` for the complete list of available 2-electron integral names

aosym [int or str] Permutation symmetry for the AO integrals

4 or '4' or 's4': 4-fold symmetry (default)
'2ij' or 's2ij' : symmetry between i, j in (ijlkl)
'2kl' or 's2kl' : symmetry between k, l in (ijlkl)
1 or '1' or 's1': no symmetry
'a4ij' : 4-fold symmetry with anti-symmetry between i, j in (ijlkl) (TODO)
'a4kl' : 4-fold symmetry with anti-symmetry between k, l in (ijlkl) (TODO)
'a2ij' : anti-symmetry between i, j in (ijlkl) (TODO)
'a2kl' : anti-symmetry between k, l in (ijlkl) (TODO)

comp [int] Components of the integrals, e.g. int2e_ip_sph has 3 components.

max_memory [float or int] The maximum size of cache to use (in MB), large cache may **not** improve performance.

ioblk_size [float or int] The block size for IO, large block size may **not** improve performance

verbose [int] Print level

compact [bool] When compact is True, depending on the four oribital sets, the returned MO integrals has (up to 4-fold) permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

Returns: None

Examples:

```
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> import h5py
>>> def view(h5file, dataname='eri_mo'):
...     f5 = h5py.File(h5file)
...     print('dataset %s, shape %s' % (str(f5.keys()), str(f5[dataname].shape)))
...     f5.close()
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> mo2 = numpy.random.random((mol.nao_nr(), 8))
>>> mo3 = numpy.random.random((mol.nao_nr(), 6))
>>> mo4 = numpy.random.random((mol.nao_nr(), 4))
>>> ao2mo.outcore.general(mol, (mo1,mo2,mo3,mo4), 'oh2.h5')
>>> view('oh2.h5')
dataset ['eri_mo'], shape (80, 24)
>>> ao2mo.outcore.general(mol, (mo1,mo2,mo3,mo3), 'oh2.h5')
>>> view('oh2.h5')
dataset ['eri_mo'], shape (80, 21)
>>> ao2mo.outcore.general(mol, (mo1,mo2,mo3,mo3), 'oh2.h5', compact=False)
>>> view('oh2.h5')
dataset ['eri_mo'], shape (80, 36)
>>> ao2mo.outcore.general(mol, (mo1,mo1,mo2,mo2), 'oh2.h5')
```

```
>>> view('oh2.h5')
dataset ['eri_mo'], shape (55, 36)
>>> ao2mo.outcore.general(mol, (mo1,mo1,mo1,mo1), 'oh2.h5', dataname='new')
>>> view('oh2.h5', 'new')
dataset ['eri_mo', 'new'], shape (55, 55)
>>> ao2mo.outcore.general(mol, (mo1,mo1,mo1,mo1), 'oh2.h5', intor='int2e_ip1_sph
 ↪', aosym='s1', comp=3)
>>> view('oh2.h5')
dataset ['eri_mo', 'new'], shape (3, 100, 100)
>>> ao2mo.outcore.general(mol, (mo1,mo1,mo1,mo1), 'oh2.h5', intor='int2e_ip1_sph
 ↪', aosym='s2kl', comp=3)
>>> view('oh2.h5')
dataset ['eri_mo', 'new'], shape (3, 100, 55)
```

pyscf.ao2mo.outcore.**general_iofree**(*mol*, *mo_coeffs*, *intor='int2e_sph'*, *aosym='s4'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *verbose=2*, *compact=True*)

For the given four sets of orbitals, transfer arbitrary spherical AO integrals to MO integrals on the fly. This function is a wrap for `ao2mo.outcore.general()`. It's not really IO free. The returned MO integrals are held in memory. For backward compatibility, it is used to replace the non-existed function direct.general_iofree.

Args:

mol [`Mole` object] AO integrals will be generated in terms of mol._atm, mol._bas, mol._env

mo_coeffs [4-item list of ndarray] Four sets of orbital coefficients, corresponding to the four indices of (ij|kl)

Kwargs

intor [str] Name of the 2-electron integral. Ref to `getints_by_shell()` for the complete list of available 2-electron integral names

aosym [int or str] Permutation symmetry for the AO integrals

4 or '4' or 's4': 4-fold symmetry (default)
'2ij' or 's2ij' : symmetry between i, j in (ij|kl)
'2kl' or 's2kl' : symmetry between k, l in (ij|kl)
1 or '1' or 's1': no symmetry
'a4ij' : 4-fold symmetry with anti-symmetry between i, j in (ij|kl) (TODO)
'a4kl' : 4-fold symmetry with anti-symmetry between k, l in (ij|kl) (TODO)
'a2ij' : anti-symmetry between i, j in (ij|kl) (TODO)
'a2kl' : anti-symmetry between k, l in (ij|kl) (TODO)

comp [int] Components of the integrals, e.g. int2e_ip_sph has 3 components.

verbose [int] Print level

compact [bool] When compact is True, depending on the four oribital sets, the returned MO integrals has (up to 4-fold) permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

Returns: 2D/3D MO-integral array. They may or may not have the permutation symmetry, depending on the given orbitals, and the kwargs compact. If the four sets of orbitals are identical, the MO integrals will at most have 4-fold symmetry.

Examples:

```
>>> from pyscf import gto
>>> from pyscf import ao2mo
>>> import h5py
>>> def view(h5file, dataname='eri_mo'):
...     f5 = h5py.File(h5file)
...     print('dataset %s, shape %s' % (str(f5.keys()), str(f5[dataname].shape)))
...     f5.close()
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> mo1 = numpy.random.random((mol.nao_nr(), 10))
>>> mo2 = numpy.random.random((mol.nao_nr(), 8))
>>> mo3 = numpy.random.random((mol.nao_nr(), 6))
>>> mo4 = numpy.random.random((mol.nao_nr(), 4))
>>> eri1 = ao2mo.outcore.general_iofree(mol, (mo1,mo2,mo3,mo4))
>>> print(eri1.shape)
(80, 24)
>>> eri1 = ao2mo.outcore.general_iofree(mol, (mo1,mo2,mo3,mo3))
>>> print(eri1.shape)
(80, 21)
>>> eri1 = ao2mo.outcore.general_iofree(mol, (mo1,mo2,mo3,mo3), compact=False)
>>> print(eri1.shape)
(80, 36)
>>> eri1 = ao2mo.outcore.general_iofree(mol, (mo1,mo1,mo1,mo1), intor='int2e_ip1_
↪sph', aosym='s1', comp=3)
>>> print(eri1.shape)
(3, 100, 100)
>>> eri1 = ao2mo.outcore.general_iofree(mol, (mo1,mo1,mo1,mo1), intor='int2e_ip1_
↪sph', aosym='s2kl', comp=3)
>>> print(eri1.shape)
(3, 100, 55)
```

pyscf.ao2mo.outcore.**half_e1**(*mol*, *mo_coeffs*, *swapfile*, *intor='int2e_sph'*, *aosym='s4'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *verbose=2*, *compact=True*, *ao2mopt=None*)

Half transform arbitrary spherical AO integrals to MO integrals for the given two sets of orbitals

Args:

mol [Mole object] AO integrals will be generated in terms of mol._atm, mol._bas, mol._env

mo_coeff [ndarray] Transform (ij|kl) with the same set of orbitals.

swapfile [str or h5py File or h5py Group object] To store the transformed integrals, in HDF5 format. The transformed integrals are saved in blocks.

Kwargs

intor [str] Name of the 2-electron integral. Ref to getints_by_shell() for the complete list of available 2-electron integral names

aosym [int or str] Permutation symmetry for the AO integrals

4 or '4' or 's4': 4-fold symmetry (default)
'2ij' or 's2ij' : symmetry between i, j in (ij|kl)
'2kl' or 's2kl' : symmetry between k, l in (ij|kl)
1 or '1' or 's1': no symmetry
'a4ij' : 4-fold symmetry with anti-symmetry between i, j in (ij|kl) (TODO)

'a4kl' : 4-fold symmetry with anti-symmetry between k, l in (ij|kl) (TODO)
'a2ij' : anti-symmetry between i, j in (ij|kl) (TODO)
'a2kl' : anti-symmetry between k, l in (ij|kl) (TODO)

comp [int] Components of the integrals, e.g. int2e_ip_sph has 3 components.

verbose [int] Print level

max_memory [float or int] The maximum size of cache to use (in MB), large cache may **not** improve performance.

ioblk_size [float or int] The block size for IO, large block size may **not** improve performance

verbose [int] Print level

compact [bool] When compact is True, depending on the four oribital sets, the returned MO integrals has (up to 4-fold) permutation symmetry. If it's False, the function will abandon any permutation symmetry, and return the "plain" MO integrals

ao2mopt [`AO2MOpt` object] Precomputed data to improve perfomance

Returns: None

1.7.4 addons

class `pyscf.ao2mo.addons.`**`load`**(*eri*, *dataname='eri_mo'*)
load 2e integrals from hdf5 file

Usage:

with load(erifile) as eri: print eri.shape

`pyscf.ao2mo.addons.`**`restore`**(*symmetry*, *eri*, *norb*, *tao=None*)
Convert the 2e integrals between different level of permutation symmetry (8-fold, 4-fold, or no symmetry)

Args:

symmetry [int or str] code to present the target symmetry of 2e integrals

's8' or '8' or 8 : 8-fold symmetry
's4' or '4' or 4 : 4-fold symmetry
's1' or '1' or 1 : no symmetry
's2ij' or '2ij' : symmetric ij pair for (ij|kl) (TODO)
's2ij' or '2kl' : symmetric kl pair for (ij|kl) (TODO)

eri [ndarray] The symmetry of eri is determined by the size of eri and norb

norb [int] The symmetry of eri is determined by the size of eri and norb

Returns: ndarray. The shape depends on the target symmetry.

8 : (norb*(norb+1)/2)*(norb*(norb+1)/2+1)/2
4 : (norb*(norb+1)/2, norb*(norb+1)/2)
1 : (norb, norb, norb, norb)

Examples:

```
>>> from pyscf import gto
>>> from pyscf.scf import _vhf
>>> from pyscf import ao2mo
>>> mol = gto.M(atom='O 0 0 0; H 0 1 0; H 0 0 1', basis='sto3g')
>>> eri = mol.intor('int2e')
>>> eri1 = ao2mo.restore(1, eri, mol.nao_nr())
>>> eri4 = ao2mo.restore(4, eri, mol.nao_nr())
>>> eri8 = ao2mo.restore(8, eri, mol.nao_nr())
>>> print(eri1.shape)
(7, 7, 7, 7)
>>> print(eri1.shape)
(28, 28)
>>> print(eri1.shape)
(406,)
```

1.8 mcscf — Multi-configurational self-consistent field

CASCI and CASSCF

Simple usage:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol).run()
>>> mc = mcscf.CASCI(mf, 6, 6)
>>> mc.kernel()[0]
-108.980200816243354
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.kernel()[0]
-109.044401882238134
>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> cas_list = [5,6,8,9] # pick orbitals for CAS space, 1-based indices
>>> mo = mcscf.sort_mo(mc, mf.mo_coeff, cas_list)
>>> mc.kernel(mo)[0]
-109.007378939813691
```

`mcscf.CASSCF()` or `mcscf.CASCI()` returns a proper instance of CASSCF/CASCI class. There are some parameters to control the CASSCF/CASCI method.

verbose [int] Print level. Default value equals to `Mole.verbose`.

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`.

ncas [int] Active space size.

nelecas [tuple of int] Active (nelec_alpha, nelec_beta)

ncore [int or tuple of int] Core electron number. In UHF-CASSCF, it's a tuple to indicate the different core eletron numbers.

natorb [bool] Whether to restore the natural orbital during CASSCF optimization. Default is not.

canonicalization [bool] Whether to canonicalize orbitals. Default is True.

fcisolver [an instance of `FCISolver`] The pyscf.fci module provides several FCISolver for different scenario. Generally, fci.direct_spin1.FCISolver can be used for all RHF-CASSCF. However, a proper FCISolver can provide better performance and better numerical stability. One can either use

`fci.solver()` function to pick the FCISolver by the program or manually assigen the FCISolver to this attribute, e.g.

```
>>> from pyscf import fci
>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> mc.fcisolver = fci.solver(mol, singlet=True)
>>> mc.fcisolver = fci.direct_spin1.FCISolver(mol)
```

You can control FCISolver by setting e.g.:

```
>>> mc.fcisolver.max_cycle = 30
>>> mc.fcisolver.conv_tol = 1e-7
```

For more details of the parameter for FCISolver, See `fci`.

By replacing this fcisolver, you can easily use the CASCI/CASSCF solver with other FCI replacements, such as DMRG, QMC. See `dmrgscf` and `fciqmcscf`.

The Following attributes are used for CASSCF

conv_tol [float] Converge threshold. Default is 1e-7

conv_tol_grad [float] Converge threshold for CI gradients and orbital rotation gradients. Default is 1e-4

max_stepsize [float] The step size for orbital rotation. Small step size is prefered. Default is 0.03. (NOTE although the default step size is small enough for many systems, it happens that the orbital optimizor crosses the barriar of local minimum and converge to the neighbour solution, e.g. the CAS(4,4) for C2H4 in the test files. In these cases, one need to fine the optimization by reducing max_stepsize, max_ci_stepsize and max_cycle_micro, max_cycle_micro_inner and ah_start_tol.)

```
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.max_stepsize = .01
>>> mc.max_cycle_micro = 1
>>> mc.max_cycle_macro = 100
>>> mc.max_cycle_micro_inner = 1
>>> mc.ah_start_tol = 1e-6
```

max_ci_stepsize [float] The max size for approximate CI updates. The approximate updates are used in 1-step algorithm, to estimate the change of CI wavefunction wrt the orbital rotation. Small step size is prefered. Default is 0.01.

max_cycle_macro [int] Max number of macro iterations. Default is 50.

max_cycle_micro [int] Max number of micro iterations in each macro iteration. Depending on systems, increasing this value might reduce the total macro iterations. Generally, 2 - 3 steps should be enough. Default is 2.

max_cycle_micro_inner [int] Max number of steps for the orbital rotations allowed for the augmented hessian solver. It can affect the actual size of orbital rotation. Even with a small max_stepsize, a few max_cycle_micro_inner can accumulate the rotation and leads to a significant change of the CAS space. Depending on systems, increasing this value migh reduce the total number of macro iterations. The value between 2 - 8 is preferred. Default is 4.

frozen [int or list] If integer is given, the inner-most orbitals are excluded from optimization. Given the orbital indices (0-based) in a list, any doubly occupied core orbitals, active orbitals and external orbitals can be frozen.

ah_level_shift [float, for AH solver.] Level shift for the Davidson diagonalization in AH solver. Default is 0.

ah_conv_tol [float, for AH solver.] converge threshold for Davidson diagonalization in AH solver. Default is 1e-8.

ah_max_cycle [float, for AH solver.] Max number of iterations allowd in AH solver. Default is 20.

ah_lindep [float, for AH solver.] Linear dependence threshold for AH solver. Default is 1e-16.

ah_start_tol [flat, for AH solver.] In AH solver, the orbital rotation is started without completely solving the AH problem. This value is to control the start point. Default is 1e-4.

ah_start_cycle [int, for AH solver.] In AH solver, the orbital rotation is started without completely solving the AH problem. This value is to control the start point. Default is 3.

`ah_conv_tol`, `ah_max_cycle`, `ah_lindep`, `ah_start_tol` and `ah_start_cycle` can affect the accuracy and performance of CASSCF solver. Lower `ah_conv_tol` and `ah_lindep` can improve the accuracy of CASSCF optimization, but slow down the performance.

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.conv_tol = 1e-10
>>> mc.ah_conv_tol = 1e-5
>>> mc.kernel()
-109.044401898486001
>>> mc.ah_conv_tol = 1e-10
>>> mc.kernel()
-109.044401887945668
```

chkfile [str] Checkpoint file to save the intermediate orbitals during the CASSCF optimization. Default is the checkpoint file of mean field object.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

ci [ndarray] CAS space FCI coefficients

converged [bool, for CASSCF only] It indicates CASSCF optimization converged or not.

mo_energy: ndarray, Diagonal elements of general Fock matrix

mo_coeff [ndarray, for CASSCF only] Optimized CASSCF orbitals coefficients Note the orbitals are NOT natural orbitals by default. There are two inbuilt methods to convert the mo_coeff to natural orbitals. 1. Set .natorb attribute. It can be used before calculation. 2. call .cas_natorb_ method after the calculation to in-place convert the orbitals

1.8.1 CASSCF active space solver

DMRG solver

FCIQMC solver

State-average FCI solver

State-average with mixed solver

1.8.2 Symmetry broken

1.8.3 Initial guess

1.8.4 Program reference

CASCI

class `pyscf.mcscf.casci.`**`CASCI`**(*mf*, *ncas*, *nelecas*, *ncore=None*)

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`.

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`.

ncas [int] Active space size.

nelecas [tuple of int] Active (nelec_alpha, nelec_beta)

ncore [int or tuple of int] Core electron number. In UHF-CASSCF, it's a tuple to indicate the different core eletron numbers.

natorb [bool] Whether to restore the natural orbital in CAS space. Default is not. Be very careful to set this parameter when CASCI/CASSCF are combined with DMRG solver because this parameter changes the orbital ordering which DMRG relies on.

canonicalization [bool] Whether to canonicalize orbitals. Default is True.

fcisolver [an instance of `FCISolver`] The pyscf.fci module provides several FCISolver for different scenario. Generally, fci.direct_spin1.FCISolver can be used for all RHF-CASSCF. However, a proper FCISolver can provide better performance and better numerical stability. One can either use `fci.solver()` function to pick the FCISolver by the program or manually assigen the FCISolver to this attribute, e.g.

```
>>> from pyscf import fci
>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> mc.fcisolver = fci.solver(mol, singlet=True)
>>> mc.fcisolver = fci.direct_spin1.FCISolver(mol)
```

You can control FCISolver by setting e.g.:

```
>>> mc.fcisolver.max_cycle = 30
>>> mc.fcisolver.conv_tol = 1e-7
```

For more details of the parameter for FCISolver, See `fci`.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

ci [ndarray] CAS space FCI coefficients

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASCI(mf, 6, 6)
>>> mc.kernel()[0]
-108.980200816243354
```

canonicalize(*mc*, *mo_coeff=None*, *ci=None*, *eris=None*, *sort=False*, *cas_natorb=False*, *casdm1=None*, *verbose=3*)
Canonicalize CASCI/CASSCF orbitals

Args: mc : a CASSCF/CASCI object or RHF object

Returns: A tuple, (natural orbitals, CI coefficients, orbital energies) The orbital energies are the diagonal terms of general Fock matrix.

canonicalize_(*mo_coeff=None*, *ci=None*, *eris=None*, *sort=False*, *cas_natorb=False*, *casdm1=None*, *verbose=None*)
Canonicalize CASCI/CASSCF orbitals

Args: mc : a CASSCF/CASCI object or RHF object

Returns: A tuple, (natural orbitals, CI coefficients, orbital energies) The orbital energies are the diagonal terms of general Fock matrix.

cas_natorb(*mo_coeff=None*, *ci=None*, *eris=None*, *sort=False*, *casdm1=None*, *verbose=None*)
Transform active orbitals to natrual orbitals, and update the CI wfn

Args: mc : a CASSCF/CASCI object or RHF object

Kwargs:

sort [bool] Sort natural orbitals wrt the occupancy. Be careful with this option since the resultant natural orbitals might have the different symmetry to the irreps indicated by CASSCF.orbsym

Returns: A tuple, the first item is natural orbitals, the second is updated CI coefficients, the third is the natural occupancy associated to the natural orbitals.

cas_natorb_(*mo_coeff=None*, *ci=None*, *eris=None*, *sort=False*, *casdm1=None*, *verbose=None*)
Transform active orbitals to natrual orbitals, and update the CI wfn

Args: mc : a CASSCF/CASCI object or RHF object

Kwargs:

sort [bool] Sort natural orbitals wrt the occupancy. Be careful with this option since the resultant natural orbitals might have the different symmetry to the irreps indicated by CASSCF.orbsym

Returns: A tuple, the first item is natural orbitals, the second is updated CI coefficients, the third is the natural occupancy associated to the natural orbitals.

fix_spin_(*shift=0.2*, *ss=None*)
Use level shift to control FCI solver spin.

$$(H + shift * S^2)|\Psi\rangle = E|\Psi\rangle$$

Kwargs:

shift [float] Level shift for states which have different spin

ss [number] S^2 expection value == s*(s+1)

get_h1cas(*mo_coeff=None*, *ncas=None*, *ncore=None*)
CAS sapce one-electron hamiltonian

Args: casci : a CASSCF/CASCI object or RHF object

Returns: A tuple, the first is the effective one-electron hamiltonian defined in CAS space, the second is the electronic energy from core.

get_h1eff(*mo_coeff=None*, *ncas=None*, *ncore=None*)
CAS sapce one-electron hamiltonian

Args: casci : a CASSCF/CASCI object or RHF object

Returns: A tuple, the first is the effective one-electron hamiltonian defined in CAS space, the second is the electronic energy from core.

h1e_for_cas(*mo_coeff=None*, *ncas=None*, *ncore=None*)
CAS sapce one-electron hamiltonian

Args: casci : a CASSCF/CASCI object or RHF object

Returns: A tuple, the first is the effective one-electron hamiltonian defined in CAS space, the second is the electronic energy from core.

make_rdm1(*mo_coeff=None*, *ci=None*, *ncas=None*, *nelecas=None*, *ncore=None*)
One-particle density matrix in AO representation

make_rdm1s(*mo_coeff=None*, *ci=None*, *ncas=None*, *nelecas=None*, *ncore=None*)
One-particle density matrices for alpha and beta spin

sort_mo(*caslst*, *mo_coeff=None*, *base=1*)
Select active space. See also *pyscf.mcscf.addons.sort_mo()*

state_average_(*weights=(0.5, 0.5)*)
State average over the energy. The energy funcitonal is E = w1<psi1|H|psi1> + w2<psi2|H|psi2> + ...

Note we may need change the FCI solver to

mc.fcisolver = fci.solver(mol, False)

before calling state_average_(mc), to mix the singlet and triplet states

state_specific_(*state=1*)
For excited state

Kwargs: state : int 0 for ground state; 1 for first excited state.

pyscf.mcscf.casci.**canonicalize**(*mc*, *mo_coeff=None*, *ci=None*, *eris=None*, *sort=False*, *cas_natorb=False*, *casdm1=None*, *verbose=3*)
Canonicalize CASCI/CASSCF orbitals

Args: mc : a CASSCF/CASCI object or RHF object

Returns: A tuple, (natural orbitals, CI coefficients, orbital energies) The orbital energies are the diagonal terms of general Fock matrix.

pyscf.mcscf.casci.**cas_natorb**(*mc*, *mo_coeff=None*, *ci=None*, *eris=None*, *sort=False*, *casdm1=None*, *verbose=None*)
Transform active orbitals to natrual orbitals, and update the CI wfn

Args: mc : a CASSCF/CASCI object or RHF object

Kwargs:

sort [bool] Sort natural orbitals wrt the occupancy. Be careful with this option since the resultant natural orbitals might have the different symmetry to the irreps indicated by CASSCF.orbsym

Returns: A tuple, the first item is natural orbitals, the second is updated CI coefficients, the third is the natural occupancy associated to the natural orbitals.

pyscf.mcscf.casci.**get_fock**(*mc*, *mo_coeff=None*, *ci=None*, *eris=None*, *casdm1=None*, *verbose=None*)
Generalized Fock matrix in AO representation

pyscf.mcscf.casci.**h1e_for_cas**(*casci*, *mo_coeff=None*, *ncas=None*, *ncore=None*)
CAS sapce one-electron hamiltonian

Args: casci : a CASSCF/CASCI object or RHF object

Returns: A tuple, the first is the effective one-electron hamiltonian defined in CAS space, the second is the electronic energy from core.

pyscf.mcscf.casci.**kernel**(*casci*, *mo_coeff=None*, *ci0=None*, *verbose=3*)
CASCI solver

pyscf.mcscf.casci_uhf.**h1e_for_cas**(*casci*, *mo_coeff=None*, *ncas=None*, *ncore=None*)
CAS sapce one-electron hamiltonian for UHF-CASCI or UHF-CASSCF

Args: casci : a U-CASSCF/U-CASCI object or UHF object

pyscf.mcscf.casci_uhf.**kernel**(*casci*, *mo_coeff=None*, *ci0=None*, *verbose=3*)
UHF-CASCI solver

CASSCF

class pyscf.mcscf.mc1step.**CASSCF**(*mf*, *ncas*, *nelecas*, *ncore=None*, *frozen=None*)

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`.

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`.

ncas [int] Active space size.

nelecas [tuple of int] Active (nelec_alpha, nelec_beta)

ncore [int or tuple of int] Core electron number. In UHF-CASSCF, it's a tuple to indicate the different core eletron numbers.

natorb [bool] Whether to restore the natural orbital in CAS space. Default is not. Be very careful to set this parameter when CASCI/CASSCF are combined with DMRG solver because this parameter changes the orbital ordering which DMRG relies on.

canonicalization [bool] Whether to canonicalize orbitals. Default is True.

fcisolver [an instance of `FCISolver`] The pyscf.fci module provides several FCISolver for different scenario. Generally, fci.direct_spin1.FCISolver can be used for all RHF-CASSCF. However, a proper FCISolver can provide better performance and better numerical stability. One can either use `fci.solver()` function to pick the FCISolver by the program or manually assigen the FCISolver to this attribute, e.g.

```
>>> from pyscf import fci
>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> mc.fcisolver = fci.solver(mol, singlet=True)
>>> mc.fcisolver = fci.direct_spin1.FCISolver(mol)
```

You can control FCISolver by setting e.g.:

```
>>> mc.fcisolver.max_cycle = 30
>>> mc.fcisolver.conv_tol = 1e-7
```

For more details of the parameter for FCISolver, See `fci`.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

ci [ndarray] CAS space FCI coefficients

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASCI(mf, 6, 6)
>>> mc.kernel()[0]
-108.980200816243354
CASSCF
```

Extra attributes for CASSCF:

conv_tol [float] Converge threshold. Default is 1e-7

conv_tol_grad [float] Converge threshold for CI gradients and orbital rotation gradients. Default is 1e-4

max_stepsize [float] The step size for orbital rotation. Small step (0.005 - 0.05) is prefered. (see notes in max_cycle_micro_inner attribute) Default is 0.03.

max_cycle_macro [int] Max number of macro iterations. Default is 50.

max_cycle_micro [int] Max number of micro iterations in each macro iteration. Depending on systems, increasing this value might reduce the total macro iterations. Generally, 2 - 5 steps should be enough. Default is 3.

ah_level_shift [float, for AH solver.] Level shift for the Davidson diagonalization in AH solver. Default is 1e-8.

ah_conv_tol [float, for AH solver.] converge threshold for AH solver. Default is 1e-12.

ah_max_cycle [float, for AH solver.] Max number of iterations allowd in AH solver. Default is 30.

ah_lindep [float, for AH solver.] Linear dependence threshold for AH solver. Default is 1e-14.

ah_start_tol [flat, for AH solver.] In AH solver, the orbital rotation is started without completely solving the AH problem. This value is to control the start point. Default is 0.2.

ah_start_cycle [int, for AH solver.] In AH solver, the orbital rotation is started without completely solving the AH problem. This value is to control the start point. Default is 2.

`ah_conv_tol`, `ah_max_cycle`, `ah_lindep`, `ah_start_tol` and `ah_start_cycle` can affect the accuracy and performance of CASSCF solver. Lower `ah_conv_tol` and `ah_lindep` might improve the accuracy of CASSCF optimization, but decrease the performance.

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
```

```
>>> mf = scf.UHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.conv_tol = 1e-10
>>> mc.ah_conv_tol = 1e-5
>>> mc.kernel()
-109.044401898486001
>>> mc.ah_conv_tol = 1e-10
>>> mc.kernel()
-109.044401887945668
```

chkfile [str] Checkpoint file to save the intermediate orbitals during the CASSCF optimization. Default is the checkpoint file of mean field object.

ci_response_space [int] subspace size to solve the CI vector response. Default is 3.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

ci [ndarray] CAS space FCI coefficients

converged [bool] It indicates CASSCF optimization converged or not.

mo_coeff [ndarray] Optimized CASSCF orbitals coefficients

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.kernel()[0]
-109.044401882238134
```

rotate_mo(*mo*, *u*, *log=None*)
Rotate orbitals with the given unitary matrix

solve_approx_ci(*h1*, *h2*, *ci0*, *ecore*, *e_ci*, *envs*)
Solve CI eigenvalue/response problem approximately

pyscf.mcscf.mc1step.**kernel**(*casscf*, *mo_coeff*, *tol=1e-07*, *conv_tol_grad=None*, *ci0=None*, *callback=None*, *verbose=3*, *dump_chk=True*)
CASSCF solver

class pyscf.mcscf.mc1step_symm.**CASSCF**(*mf*, *ncas*, *nelecas*, *ncore=None*, *frozen=None*)

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`.

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`.

ncas [int] Active space size.

nelecas [tuple of int] Active (nelec_alpha, nelec_beta)

ncore [int or tuple of int] Core electron number. In UHF-CASSCF, it's a tuple to indicate the different core eletron numbers.

natorb [bool] Whether to restore the natural orbital in CAS space. Default is not. Be very careful to set this parameter when CASCI/CASSCF are combined with DMRG solver because this parameter changes the orbital ordering which DMRG relies on.

canonicalization [bool] Whether to canonicalize orbitals. Default is True.

fcisolver [an instance of `FCISolver`] The pyscf.fci module provides several FCISolver for different scenario. Generally, fci.direct_spin1.FCISolver can be used for all RHF-CASSCF. However, a proper FCISolver can provide better performance and better numerical stability. One can either use `fci.solver()` function to pick the FCISolver by the program or manually assigen the FCISolver to this attribute, e.g.

```
>>> from pyscf import fci
>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> mc.fcisolver = fci.solver(mol, singlet=True)
>>> mc.fcisolver = fci.direct_spin1.FCISolver(mol)
```

You can control FCISolver by setting e.g.:

```
>>> mc.fcisolver.max_cycle = 30
>>> mc.fcisolver.conv_tol = 1e-7
```

For more details of the parameter for FCISolver, See `fci`.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

ci [ndarray] CAS space FCI coefficients

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASCI(mf, 6, 6)
>>> mc.kernel()[0]
-108.980200816243354
CASSCF
```

Extra attributes for CASSCF:

conv_tol [float] Converge threshold. Default is 1e-7

conv_tol_grad [float] Converge threshold for CI gradients and orbital rotation gradients. Default is 1e-4

max_stepsize [float] The step size for orbital rotation. Small step (0.005 - 0.05) is prefered. (see notes in max_cycle_micro_inner attribute) Default is 0.03.

max_cycle_macro [int] Max number of macro iterations. Default is 50.

max_cycle_micro [int] Max number of micro iterations in each macro iteration. Depending on systems, increasing this value might reduce the total macro iterations. Generally, 2 - 5 steps should be enough. Default is 3.

ah_level_shift [float, for AH solver.] Level shift for the Davidson diagonalization in AH solver. Default is 1e-8.

ah_conv_tol [float, for AH solver.] converge threshold for AH solver. Default is 1e-12.

ah_max_cycle [float, for AH solver.] Max number of iterations allowd in AH solver. Default is 30.

ah_lindep [float, for AH solver.] Linear dependence threshold for AH solver. Default is 1e-14.

ah_start_tol [flat, for AH solver.] In AH solver, the orbital rotation is started without completely solving the AH problem. This value is to control the start point. Default is 0.2.

ah_start_cycle [int, for AH solver.] In AH solver, the orbital rotation is started without completely solving the AH problem. This value is to control the start point. Default is 2.

`ah_conv_tol`, `ah_max_cycle`, `ah_lindep`, `ah_start_tol` and `ah_start_cycle` can affect the accuracy and performance of CASSCF solver. Lower `ah_conv_tol` and `ah_lindep` might improve the accuracy of CASSCF optimization, but decrease the performance.

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.conv_tol = 1e-10
>>> mc.ah_conv_tol = 1e-5
>>> mc.kernel()
-109.044401898486001
>>> mc.ah_conv_tol = 1e-10
>>> mc.kernel()
-109.044401887945668
```

chkfile [str] Checkpoint file to save the intermediate orbitals during the CASSCF optimization. Default is the checkpoint file of mean field object.

ci_response_space [int] subspace size to solve the CI vector response. Default is 3.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function `locals()`, so that the callback function can access all local variables in the current envrionment.

Saved results

e_tot [float] Total MCSCF energy (electronic energy plus nuclear repulsion)

ci [ndarray] CAS space FCI coefficients

converged [bool] It indicates CASSCF optimization converged or not.

mo_coeff [ndarray] Optimized CASSCF orbitals coefficients

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> mc.kernel()[0]
-109.044401882238134
```

addons

pyscf.mcscf.addons.**cas_natorb**(*casscf*, *mo_coeff=None*, *ci=None*, *sort=False*)
Natrual orbitals in CAS space

pyscf.mcscf.addons.**caslst_by_irrep**(*casscf*, *mo_coeff*, *cas_irrep_nocc*, *cas_irrep_ncore=None*, *s=None*, *base=1*)
Given number of active orbitals for each irrep, return the orbital indices of active space

Args: casscf : an CASSCF or CASCI object

cas_irrep_nocc [list or dict] Number of active orbitals for each irrep. It can be a dict, eg {'A1': 2, 'B2': 4} to indicate the active space size based on irrep names, or {0: 2, 3: 4} for irrep Id, or a list [2, 0, 0, 4] (identical to {0: 2, 3: 4}) in which the list index is served as the irrep Id.

Kwargs:

cas_irrep_ncore [list or dict] Number of closed shells for each irrep. It can be a dict, eg {'A1': 6, 'B2': 4} to indicate the closed shells based on irrep names, or {0: 6, 3: 4} for irrep Id, or a list [6, 0, 0, 4] (identical to {0: 6, 3: 4}) in which the list index is served as the irrep Id. If cas_irrep_ncore is not given, the program will generate a guess based on the lowest CASCI.ncore orbitals.

s [ndarray] overlap matrix

base [int] 0-based (C-like) or 1-based (Fortran-like) caslst

Returns: A list of orbital indices

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvtz', symmetry=True, verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.kernel()
>>> mc = mcscf.CASSCF(mf, 12, 4)
>>> mcscf.caslst_by_irrep(mc, mf.mo_coeff, {'E1gx':4, 'E1gy':4, 'E1ux':2, 'E1uy':
↪2})
[5, 7, 8, 10, 11, 14, 15, 20, 25, 26, 31, 32]
```

pyscf.mcscf.addons.**get_fock**(*casscf*, *mo_coeff=None*, *ci=None*)
Generalized Fock matrix in AO representation

pyscf.mcscf.addons.**hot_tuning_**(*casscf*, *configfile=None*)
Allow you to tune CASSCF parameters on the runtime

pyscf.mcscf.addons.**make_rdm1**(*casscf*, *mo_coeff=None*, *ci=None*)
One-particle densit matrix in AO representation

Args: casscf : an CASSCF or CASCI object

Kwargs:

ci [ndarray] CAS space FCI coefficients. If not given, take casscf.ci.

mo_coeff [ndarray] Orbital coefficients. If not given, take casscf.mo_coeff.

Examples:

```
>>> import scipy.linalg
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='sto-3g', verbose=0)
>>> mf = scf.RHF(mol)
>>> res = mf.scf()
```

```
>>> mc = mcscf.CASSCF(mf, 6, 6)
>>> res = mc.kernel()
>>> natocc = numpy.linalg.eigh(mcscf.make_rdm1(mc), mf.get_ovlp(), type=2)[0]
>>> print(natocc)
[ 0.0121563   0.0494735   0.0494735   1.95040395  1.95040395  1.98808879
  2.          2.          2.          2.        ]
```

pyscf.mcscf.addons.**make_rdm1s**(*casscf*, *mo_coeff=None*, *ci=None*)
Alpha and beta one-particle densit matrices in AO representation

pyscf.mcscf.addons.**map2hf**(*casscf*, *mf_mo=None*, *base=1*, *tol=0.5*)
The overlap between the CASSCF optimized orbitals and the canonical HF orbitals.

pyscf.mcscf.addons.**project_init_guess**(*casscf*, *init_mo*, *prev_mol=None*)
Project the given initial guess to the current CASSCF problem. The projected initial guess has two parts. The core orbitals are directly taken from the Hartree-Fock orbitals, and the active orbitals are projected from the given initial guess.

Args: casscf : an CASSCF or CASCI object

init_mo [ndarray or list of ndarray] Initial guess orbitals which are not orth-normal for the current molecule. When the casscf is UHF-CASSCF, the init_mo needs to be a list of two ndarrays, for alpha and beta orbitals

Kwargs:

prev_mol [an instance of Mole] If given, the inital guess orbitals are associated to the geometry and basis of prev_mol. Otherwise, the orbitals are based of the geometry and basis of casscf.mol

Returns: New orthogonal initial guess orbitals with the core taken from Hartree-Fock orbitals and projected active space from original initial guess orbitals

Examples:

```
import numpy
from pyscf import gto, scf, mcscf
mol = gto.Mole()
mol.build(atom='H 0 0 0; F 0 0 0.8', basis='ccpvdz', verbose=0)
mf = scf.RHF(mol)
mf.scf()
mc = mcscf.CASSCF(mf, 6, 6)
mo = mcscf.sort_mo(mc, mf.mo_coeff, [3,4,5,6,8,9])
print('E(0.8) = %.12f' % mc.kernel(mo)[0])
init_mo = mc.mo_coeff
for b in numpy.arange(1.0, 3., .2):
    mol.atom = [['H', (0, 0, 0)], ['F', (0, 0, b)]]
    mol.build(0, 0)
    mf = scf.RHF(mol)
    mf.scf()
    mc = mcscf.CASSCF(mf, 6, 6)
    mo = mcscf.project_init_guess(mc, init_mo)
    print('E(%2.1f) = %.12f' % (b, mc.kernel(mo)[0]))
    init_mo = mc.mo_coeff
```

pyscf.mcscf.addons.**sort_mo**(*casscf*, *mo_coeff*, *caslst*, *base=1*)
Pick orbitals for CAS space

Args: casscf : an CASSCF or CASCI object

mo_coeff [ndarray or a list of ndarray] Orbitals for CASSCF initial guess. In the UHF-CASSCF, it's a list of two orbitals, for alpha and beta spin.

caslst [list of int or nested list of int] A list of orbital indices to represent the CAS space. In the UHF-CASSCF, it's consist of two lists, for alpha and beta spin.

Kwargs:

base [int] 0-based (C-style) or 1-based (Fortran-style) caslst

Returns: An reoreded mo_coeff, which put the orbitals given by caslst in the CAS space

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.scf()
>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> cas_list = [5,6,8,9] # pi orbitals
>>> mo = mc.sort_mo(cas_list)
>>> mc.kernel(mo)[0]
-109.007378939813691
```

pyscf.mcscf.addons.**sort_mo_by_irrep**(*casscf*, *mo_coeff*, *cas_irrep_nocc*, *cas_irrep_ncore=None*, *s=None*)

Given number of active orbitals for each irrep, construct the mo initial guess for CASSCF

Args: casscf : an `CASSCF` or `CASCI` object

cas_irrep_nocc [list or dict] Number of active orbitals for each irrep. It can be a dict, eg {'A1': 2, 'B2': 4} to indicate the active space size based on irrep names, or {0: 2, 3: 4} for irrep Id, or a list [2, 0, 0, 4] (identical to {0: 2, 3: 4}) in which the list index is served as the irrep Id.

Kwargs:

cas_irrep_ncore [list or dict] Number of closed shells for each irrep. It can be a dict, eg {'A1': 6, 'B2': 4} to indicate the closed shells based on irrep names, or {0: 6, 3: 4} for irrep Id, or a list [6, 0, 0, 4] (identical to {0: 6, 3: 4}) in which the list index is served as the irrep Id. If cas_irrep_ncore is not given, the program will generate a guess based on the lowest `CASCI.ncore` orbitals.

s [ndarray] overlap matrix

Returns: sorted orbitals, ordered as [c,..,c,a,..,a,v,..,v]

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvtz', symmetry=True, verbose=0)
>>> mf = scf.RHF(mol)
>>> mf.kernel()
>>> mc = mcscf.CASSCF(mf, 12, 4)
>>> mo = mc.sort_mo_by_irrep({'E1gx':4, 'E1gy':4, 'E1ux':2, 'E1uy':2})
>>> # Same to mo = sort_mo_by_irrep(mc, mf.mo_coeff, {2: 4, 3: 4, 6: 2, 7: 2})
>>> # Same to mo = sort_mo_by_irrep(mc, mf.mo_coeff, [0, 0, 4, 4, 0, 0, 2, 2])
>>> mc.kernel(mo)[0]
-108.162863845084
```

pyscf.mcscf.addons.**spin_square**(*casscf*, *mo_coeff=None*, *ci=None*, *ovlp=None*)

Spin square of the UHF-CASSCF wavefunction

Returns: A list of two floats. The first is the expectation value of S^2. The second is the corresponding 2S+1

Examples:

```
>>> from pyscf import gto, scf, mcscf
>>> mol = gto.M(atom='O 0 0 0; O 0 0 1', basis='sto-3g', spin=2, verbose=0)
>>> mf = scf.UHF(mol)
>>> res = mf.scf()
>>> mc = mcscf.CASSCF(mf, 4, 6)
>>> res = mc.kernel()
>>> print('S^2 = %.7f, 2S+1 = %.7f' % mcscf.spin_square(mc))
S^2 = 3.9831589, 2S+1 = 4.1149284
```

pyscf.mcscf.addons.**state_average**(*casscf*, *weights=(0.5, 0.5)*)
State average over the energy. The energy funcitonal is E = w1<psi1|H|psi1> + w2<psi2|H|psi2> + ...

Note we may need change the FCI solver to

mc.fcisolver = fci.solver(mol, False)

before calling state_average_(mc), to mix the singlet and triplet states

pyscf.mcscf.addons.**state_average_**(*casscf*, *weights=(0.5, 0.5)*)
State average over the energy. The energy funcitonal is E = w1<psi1|H|psi1> + w2<psi2|H|psi2> + ...

Note we may need change the FCI solver to

mc.fcisolver = fci.solver(mol, False)

before calling state_average_(mc), to mix the singlet and triplet states

pyscf.mcscf.addons.**state_average_mix**(*casscf*, *fcisolvers*, *weights=(0.5, 0.5)*)
State-average CASSCF over multiple FCI solvers.

pyscf.mcscf.addons.**state_average_mix_**(*casscf*, *fcisolvers*, *weights=(0.5, 0.5)*)
State-average CASSCF over multiple FCI solvers.

pyscf.mcscf.addons.**state_specific**(*casscf*, *state=1*)
For excited state

Kwargs: state : int 0 for ground state; 1 for first excited state.

pyscf.mcscf.addons.**state_specific_**(*casscf*, *state=1*)
For excited state

Kwargs: state : int 0 for ground state; 1 for first excited state.

1.9 fci — Full configuration interaction

Different FCI solvers are implemented to support different type of symmetry. Symmetry

File Point group Spin singlet Real hermitian* Alpha/beta degeneracy direct_spin0_symm Yes Yes Yes Yes direct_spin1_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes No direct_nosym No No No** Yes

- Real hermitian Hamiltonian implies (ij|kl) = (ji|kl) = (ij|lk) = (ji|lk)

** Hamiltonian is real but not hermitian, (ij|kl) != (ji|kl) ...

1.9.1 direct CI

Full CI solver for spin-free Hamiltonian. This solver can be used to compute doublet, triplet,...

The CI wfn are stored as a 2D array [alpha,beta], where each row corresponds to an alpha string. For each row (alpha string), there are total-num-beta-strings of columns. Each column corresponds to a beta string.

Different FCI solvers are implemented to support different type of symmetry. Symmetry

File Point group Spin singlet Real hermitian* Alpha/beta degeneracy direct_spin0_symm Yes Yes Yes Yes direct_spin1_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes No direct_nosym No No No** Yes

- Real hermitian Hamiltonian implies (ij|kl) = (ji|kl) = (ij|lk) = (ji|lk)

** Hamiltonian is real but not hermitian, (ij|kl) != (ji|kl) ...

pyscf.fci.direct_spin1.**FCI**
alias of *FCISolver*

class pyscf.fci.direct_spin1.**FCISolver**(*mol=None*)
Full CI solver

Attributes:

verbose [int] Print level. Default value equals to Mole.verbose.

max_cycle [int] Total number of iterations. Default is 100

max_space [tuple of int] Davidson iteration space size. Default is 14.

conv_tol [float] Energy convergence tolerance. Default is 1e-10.

level_shift [float] Level shift applied in the preconditioner to avoid singularity. Default is 1e-3

davidson_only [bool] By default, the entire Hamiltonian matrix will be constructed and diagonalized if the system is small (see attribute pspace_size). Setting this parameter to True will enforce the eigenvalue problems being solved by Davidson subspace algorithm. This flag should be enabled when initial guess is given or particular spin symmetry or point-group symmetry is required because the initial guess or symmetry are completely ignored in the direct diagonlization.

pspace_size [int] The dimension of Hamiltonian matrix over which Davidson iteration algorithm will be used for the eigenvalue problem. Default is 400. This is roughly corresponding to a (6e,6o) system.

nroots [int] Number of states to be solved. Default is 1, the ground state.

spin [int or None] Spin (2S = nalpha-nbeta) of the system. If this attribute is None, spin will be determined by the argument nelec (number of electrons) of the kernel function.

wfnsym [str or int] Symmetry of wavefunction. It is used only in direct_spin1_symm and direct_spin0_symm solver.

Saved results

converged [bool] Whether davidson iteration is converged

Examples:

```
>>> from pyscf import gto, scf, ao2mo, fci
>>> mol = gto.M(atom='Li 0 0 0; Li 0 0 1', basis='sto-3g')
>>> mf = scf.RHF(mol).run()
>>> h1 = mf.mo_coeff.T.dot(mf.get_hcore()).dot(mf.mo_coeff)
>>> eri = ao2mo.kernel(mol, mf.mo_coeff)
>>> cisolver = fci.direct_spin1.FCI(mol)
>>> e, ci = cisolver.kernel(h1, eri, h1.shape[1], mol.nelec, ecore=mol.energy_
↪nuc())
>>> print(e)
-14.4197890826
```

absorb_h1e(*h1e*, *eri*, *norb*, *nelec*, *fac=1*)
Modify 2e Hamiltonian to include 1e Hamiltonian contribution.

contract_1e(*f1e*, *fcivec*, *norb*, *nelec*, *link_index=None*, ***kwargs*)
Contract the 1-electron Hamiltonian with a FCI vector to get a new FCI vector.

contract_2e(*eri*, *fcivec*, *norb*, *nelec*, *link_index=None*, ***kwargs*)
Contract the 2-electron Hamiltonian with a FCI vector to get a new FCI vector.

Note the input arg eri is NOT the 2e hamiltonian matrix, the 2e hamiltonian is

$$\begin{aligned} h2e &= eri_{pq,rs} p^+ q r^+ s \\ &= (pq|rs) p^+ r^+ s q - (pq|rs)\delta_{qr} p^+ s \end{aligned}$$

So eri is defined as

$$eri_{pq,rs} = (pq|rs) - (1/Nelec)\sum_q (pq|qs)$$

to restore the symmetry between pq and rs,

$$eri_{pq,rs} = (pq|rs) - (.5/Nelec)[\sum_q (pq|qs) + \sum_p (pq|rp)]$$

See also `direct_spin1.absorb_h1e()`

energy(*h1e*, *eri*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Compute the FCI electronic energy for given Hamiltonian and FCI vector.

get_init_guess(*norb*, *nelec*, *nroots*, *hdiag*)
Initial guess is the single Slater determinant

make_hdiag(*h1e*, *eri*, *norb*, *nelec*)
Diagonal Hamiltonian for Davidson preconditioner

make_rdm1(*fcivec*, *norb*, *nelec*, *link_index=None*)
spin-traced 1-particle density matrix

make_rdm12(*fcivec*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced 1- and 2-particle density matrices,

NOTE the 2pdm is $\langle p^\dagger q^\dagger sr\rangle$ but is stored as [p,r,q,s]

make_rdm12s(*fcivec*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin searated 1- and 2-particle density matrices, (alpha,beta) for 1-particle density matrices. (alpha,alpha,alpha,alpha), (alpha,alpha,beta,beta), (beta,beta,beta,beta) for 2-particle density matrices.

NOTE the 2pdm is $\langle p^\dagger q^\dagger sr\rangle$ but is stored as [p,r,q,s]

make_rdm1s(*fcivec*, *norb*, *nelec*, *link_index=None*)
Spin searated 1-particle density matrices, (alpha,beta)

make_rdm2(*fcivec*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced 2-particle density matrice

NOTE the 2pdm is $\langle p^\dagger q^\dagger sr\rangle$ but stored as [p,r,q,s]

pspace(*h1e*, *eri*, *norb*, *nelec*, *hdiag=None*, *np=400*)
pspace Hamiltonian to improve Davidson preconditioner. See, CPL, 169, 463

spin_square(*fcivec*, *norb*, *nelec*)
Spin square for RHF-FCI CI wfn only (obtained from spin-degenerated Hamiltonian)

trans_rdm1 (*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*)
Spin traced transition 1-particle density matrices

trans_rdm12 (*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced transition 1- and 2-particle density matrices.

trans_rdm12s (*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin separated transition 1- and 2-particle density matrices.

trans_rdm1s (*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*)
Spin separated transition 1-particle density matrices

pyscf.fci.direct_spin1.**absorb_h1e** (*h1e*, *eri*, *norb*, *nelec*, *fac=1*)
Modify 2e Hamiltonian to include 1e Hamiltonian contribution.

pyscf.fci.direct_spin1.**contract_1e** (*f1e*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Contract the 1-electron Hamiltonian with a FCI vector to get a new FCI vector.

pyscf.fci.direct_spin1.**contract_2e** (*eri*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Contract the 2-electron Hamiltonian with a FCI vector to get a new FCI vector.

Note the input arg eri is NOT the 2e hamiltonian matrix, the 2e hamiltonian is

$$\begin{aligned} h2e &= eri_{pq,rs} p^+ q r^+ s \\ &= (pq|rs) p^+ r^+ s q - (pq|rs)\delta_{qr} p^+ s \end{aligned}$$

So eri is defined as

$$eri_{pq,rs} = (pq|rs) - (1/Nelec)\sum_q (pq|qs)$$

to restore the symmetry between pq and rs,

$$eri_{pq,rs} = (pq|rs) - (.5/Nelec)[\sum_q (pq|qs) + \sum_p (pq|rp)]$$

See also `direct_spin1.absorb_h1e()`

pyscf.fci.direct_spin1.**energy** (*h1e*, *eri*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Compute the FCI electronic energy for given Hamiltonian and FCI vector.

pyscf.fci.direct_spin1.**get_init_guess** (*norb*, *nelec*, *nroots*, *hdiag*)
Initial guess is the single Slater determinant

pyscf.fci.direct_spin1.**make_hdiag** (*h1e*, *eri*, *norb*, *nelec*)
Diagonal Hamiltonian for Davidson preconditioner

pyscf.fci.direct_spin1.**make_rdm1** (*fcivec*, *norb*, *nelec*, *link_index=None*)
spin-traced 1-particle density matrix

pyscf.fci.direct_spin1.**make_rdm12** (*fcivec*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced 1- and 2-particle density matrices,

NOTE the 2pdm is $\langle p^\dagger q^\dagger s r\rangle$ but is stored as [p,r,q,s]

pyscf.fci.direct_spin1.**make_rdm12s** (*fcivec*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin searated 1- and 2-particle density matrices, (alpha,beta) for 1-particle density matrices. (alpha,alpha,alpha,alpha), (alpha,alpha,beta,beta), (beta,beta,beta,beta) for 2-particle density matrices.

NOTE the 2pdm is $\langle p^\dagger q^\dagger s r\rangle$ but is stored as [p,r,q,s]

pyscf.fci.direct_spin1.**make_rdm1s** (*fcivec*, *norb*, *nelec*, *link_index=None*)
Spin searated 1-particle density matrices, (alpha,beta)

pyscf.fci.direct_spin1.**pspace**(*h1e*, *eri*, *norb*, *nelec*, *hdiag=None*, *np=400*)
pspace Hamiltonian to improve Davidson preconditioner. See, CPL, 169, 463

pyscf.fci.direct_spin1.**trans_rdm1**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*)
Spin traced transition 1-particle density matrices

pyscf.fci.direct_spin1.**trans_rdm12**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced transition 1- and 2-particle density matrices.

pyscf.fci.direct_spin1.**trans_rdm12s**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin separated transition 1- and 2-particle density matrices.

pyscf.fci.direct_spin1.**trans_rdm1s**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*)
Spin separated transition 1-particle density matrices

Different FCI solvers are implemented to support different type of symmetry. Symmetry

File Point group Spin singlet Real hermitian* Alpha/beta degeneracy direct_spin0_symm Yes Yes Yes Yes direct_spin1_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes No direct_nosym No No No** Yes

- Real hermitian Hamiltonian implies (ij|kl) = (ji|kl) = (ij|lk) = (ji|lk)

** Hamiltonian is real but not hermitian, (ij|kl) != (ji|kl) ... FCI solver for Singlet state

Different FCI solvers are implemented to support different type of symmetry. Symmetry

File Point group Spin singlet Real hermitian* Alpha/beta degeneracy direct_spin0_symm Yes Yes Yes Yes direct_spin1_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes No direct_nosym No No No** Yes

- Real hermitian Hamiltonian implies (ij|kl) = (ji|kl) = (ij|lk) = (ji|lk)

** Hamiltonian is real but not hermitian, (ij|kl) != (ji|kl) ...

pyscf.fci.direct_spin0.**contract_1e**(*f1e*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Contract the 1-electron Hamiltonian with a FCI vector to get a new FCI vector.

pyscf.fci.direct_spin0.**contract_2e**(*eri*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Contract the 2-electron Hamiltonian with a FCI vector to get a new FCI vector.

Note the input arg eri is NOT the 2e hamiltonian matrix, the 2e hamiltonian is

$$\begin{aligned} h2e &= eri_{pq,rs} p^+ q r^+ s \\ &= (pq|rs) p^+ r^+ s q - (pq|rs)\delta_{qr} p^+ s \end{aligned}$$

So eri is defined as

$$eri_{pq,rs} = (pq|rs) - (1/Nelec)\sum_q (pq|qs)$$

to restore the symmetry between pq and rs,

$$eri_{pq,rs} = (pq|rs) - (.5/Nelec)[\sum_q (pq|qs) + \sum_p (pq|rp)]$$

See also `direct_spin1.absorb_h1e()`

pyscf.fci.direct_spin0.**make_hdiag**(*h1e*, *eri*, *norb*, *nelec*)
Diagonal Hamiltonian for Davidson preconditioner

pyscf.fci.direct_spin0.**make_rdm1**(*fcivec*, *norb*, *nelec*, *link_index=None*)
spin-traced 1-particle density matrix

pyscf.fci.direct_spin0.**make_rdm12**(*fcivec*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced 1- and 2-particle density matrices,

NOTE the 2pdm is $\langle p^\dagger q^\dagger sr\rangle$ but is stored as [p,r,q,s]

pyscf.fci.direct_spin0.**make_rdm1s**(*fcivec*, *norb*, *nelec*, *link_index=None*)
Spin searated 1-particle density matrices, (alpha,beta)

pyscf.fci.direct_spin0.**trans_rdm1**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*)
Spin traced transition 1-particle density matrices

pyscf.fci.direct_spin0.**trans_rdm12**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*, *reorder=True*)
Spin traced transition 1- and 2-particle density matrices.

pyscf.fci.direct_spin0.**trans_rdm1s**(*cibra*, *ciket*, *norb*, *nelec*, *link_index=None*)
Spin separated transition 1-particle density matrices

Different FCI solvers are implemented to support different type of symmetry. Symmetry

File Point group Spin singlet Real hermitian* Alpha/beta degeneracy direct_spin0_symm Yes Yes Yes Yes direct_spin1_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes No direct_nosym No No No** Yes

- Real hermitian Hamiltonian implies (ij|kl) = (ji|kl) = (ij|lk) = (ji|lk)

** Hamiltonian is real but not hermitian, (ij|kl) != (ji|kl) ...

Different FCI solvers are implemented to support different type of symmetry. Symmetry

File Point group Spin singlet Real hermitian* Alpha/beta degeneracy direct_spin0_symm Yes Yes Yes Yes direct_spin1_symm Yes No Yes Yes direct_spin0 No Yes Yes Yes direct_spin1 No No Yes Yes direct_uhf No No Yes No direct_nosym No No No** Yes

- Real hermitian Hamiltonian implies (ij|kl) = (ji|kl) = (ij|lk) = (ji|lk)

** Hamiltonian is real but not hermitian, (ij|kl) != (ji|kl) ...

1.9.2 cistring

pyscf.fci.cistring.**addr2str**(*norb*, *nelec*, *addr*)
Convert CI determinant address to string

pyscf.fci.cistring.**addrs2str**(*norb*, *nelec*, *addrs*)
Convert a list of CI determinant address to string

pyscf.fci.cistring.**gen_cre_str_index**(*orb_list*, *nelec*)
linkstr_index to map between N electron string to N+1 electron string. It maps the given string to the address of the string which is generated by the creation operator.

For given string str0, index[str0] is nvir x 4 array. Each entry [i(cre),–,str1,sign] means starting from str0, creating i, to get str1.

pyscf.fci.cistring.**gen_des_str_index**(*orb_list*, *nelec*)
linkstr_index to map between N electron string to N-1 electron string. It maps the given string to the address of the string which is generated by the annihilation operator.

For given string str0, index[str0] is nvir x 4 array. Each entry [–,i(des),str1,sign] means starting from str0, annihilating i, to get str1.

pyscf.fci.cistring.**gen_linkstr_index**(*orb_list*, *nocc*, *strs=None*, *tril=False*)
Look up table, for the strings relationship in terms of a creation-annihilating operator pair.

For given string str0, index[str0] is (nocc+nocc*nvir) x 4 array. The first nocc rows [i(:occ),i(:occ),str0,sign] are occupied-occupied excitations, which do not change the string. The next nocc*nvir rows [a(:vir),i(:occ),str1,sign] are occupied-virtual exciations, starting from str0, annihilating i, creating a, to get str1.

pyscf.fci.cistring.**gen_linkstr_index_trilidx**(*orb_list*, *nocc*, *strs=None*)
Generate linkstr_index with the assumption that $p^+q|0\rangle$ where $p > q$. So the resultant link_index has the structure `[pq,*,str1,sign]`. It is identical to a call to `reform_linkstr_index(gen_linkstr_index(...))`.

pyscf.fci.cistring.**gen_strings4orblist**(*orb_list*, *nelec*)
Generate string from the given orbital list.

Returns: list of int64. One int64 element represents one string in binary format. The binary format takes the convention that the one bit stands for one orbital, bit-1 means occupied and bit-0 means unoccupied. The lowest (right-most) bit corresponds to the lowest orbital in the orb_list.

Exampels:

```
>>> [bin(x) for x in gen_strings4orblist((0,1,2,3),2)]
[0b11, 0b101, 0b110, 0b1001, 0b1010, 0b1100]
>>> [bin(x) for x in gen_strings4orblist((3,1,0,2),2)]
[0b1010, 0b1001, 0b11, 0b1100, 0b110, 0b101]
```

pyscf.fci.cistring.**reform_linkstr_index**(*link_index*)
Compress the (a, i) pair index in linkstr_index to a lower triangular index, to match the 4-fold symmetry of integrals.

pyscf.fci.cistring.**str2addr**(*norb*, *nelec*, *string*)
Convert string to CI determinant address

pyscf.fci.cistring.**strs2addr**(*norb*, *nelec*, *strings*)
Convert a list of string to CI determinant address

pyscf.fci.cistring.**tn_strs**(*norb*, *nelec*, *n*)
Generate strings for Tn amplitudes. Eg n=1 (T1) has nvir*nocc strings, n=2 (T2) has nvir*(nvir-1)/2 * nocc*(nocc-1)/2 strings.

1.9.3 spin operator

pyscf.fci.spin_op.**contract_ss**(*fcivec*, *norb*, *nelec*)
Contract spin square operator with FCI wavefunction $S^2|CI>$

pyscf.fci.spin_op.**local_spin**(*fcivec*, *norb*, *nelec*, *mo_coeff=None*, *ovlp=1*, *aolst=[]*)
Local spin expectation value, which is defined as

<CI|ao><ao|S^2|CI>

For a complete list of AOs, I = sum |ao><ao|, it becomes <CI|S^2|CI>

pyscf.fci.spin_op.**spin_square**(*fcivec*, *norb*, *nelec*, *mo_coeff=None*, *ovlp=1*)
General spin square operator.

... math:

```
<CI|S_+*S_-|CI> &= n_\alpha + \delta_{ik}\delta_{jl}Gamma_{i\alpha k\beta ,j\beta
↪l\alpha } \\
<CI|S_-*S_+|CI> &= n_\beta + \delta_{ik}\delta_{jl}Gamma_{i\beta k\alpha ,j\alpha
↪l\beta } \\
<CI|S_z*S_z|CI> &= \delta_{ik}\delta_{jl}(Gamma_{i\alpha k\alpha ,j\alpha l\alpha
↪}
```

```
                - Gamma_{i\alpha k\alpha ,j\beta l\beta }
                - Gamma_{i\beta k\beta ,j\alpha l\alpha}
                + Gamma_{i\beta k\beta ,j\beta l\beta})
                + (n_\alpha+n_\beta)/4
```

Given the overlap betwen non-degenerate alpha and beta orbitals, this function can compute the expectation value spin square operator for UHF-FCI wavefunction

pyscf.fci.spin_op.**spin_square0**(*fcivec*, *norb*, *nelec*)
Spin square for RHF-FCI CI wfn only (obtained from spin-degenerated Hamiltonian)

1.9.4 rdm

FCI 1, 2, 3, 4-particle density matrices.

pyscf.fci.rdm.**make_dm123**(*fname*, *cibra*, *ciket*, *norb*, *nelec*)
Spin traced 1, 2 and 3-particle density matrices.

Note: In this function, 2pdm is $\langle p^\dagger q r^\dagger s\rangle$; 3pdm is $\langle p^\dagger q r^\dagger s t^\dagger u\rangle$. After calling reorder_dm123, the 2pdm and 3pdm are transformed to standard definition: 2pdm = $\langle p^\dagger q^\dagger r s\rangle$ but is stored as [p,s,q,r]; 3pdm = $\langle p^\dagger q^\dagger r^\dagger s t u\rangle$, stored as [p,u,q,t,r,s].

---

pyscf.fci.rdm.**make_dm1234**(*fname*, *cibra*, *ciket*, *norb*, *nelec*)
Spin traced 1, 2, 3 and 4-particle density matrices.

Note: In this function, 2pdm is $\langle p^\dagger q r^\dagger s\rangle$; 3pdm is $\langle p^\dagger q r^\dagger s t^\dagger u\rangle$; 4pdm is $\langle p^\dagger q r^\dagger s t^\dagger u v^\dagger w\rangle$. After calling reorder_dm1234, the 2pdm and 3pdm and 4pdm are transformed to standard definition: 2pdm = $\langle p^\dagger q^\dagger s r\rangle$ but is stored as [p,r,q,s]; 3pdm = $\langle p^\dagger q^\dagger r^\dagger u t s\rangle$, stored as [p,s,q,t,r,u]; 4pdm = $\langle p^\dagger q^\dagger r^\dagger s^d a g g e r w v u t\rangle$, stored as [p,t,q,u,r,v,s,w].

1.9.5 addons

pyscf.fci.addons.**cre_a**(*ci0*, *norb*, *neleca_nelecb*, *ap_id*)
Construct (N+1)-electron wavefunction by adding an alpha electron in the N-electron wavefunction.

... math:

```
|N+1\rangle = \hat{a}^+_p |N\rangle
```

Args:

ci0 [2D array] CI coefficients, row for alpha strings and column for beta strings.

norb [int] Number of orbitals.

(neleca,nelecb) [(int,int)] Number of (alpha, beta) electrons of the input CI function

ap_id [int] Orbital index (0-based), for the creation operator

Returns: 2D array, row for alpha strings and column for beta strings. Note it has different number of rows to the input CI coefficients.

pyscf.fci.addons.**cre_b**(*ci0*, *norb*, *neleca_nelecb*, *ap_id*)
Construct (N+1)-electron wavefunction by adding a beta electron in the N-electron wavefunction.

Args:

ci0 [2D array] CI coefficients, row for alpha strings and column for beta strings.

norb [int] Number of orbitals.

(neleca,nelecb) [(int,int)] Number of (alpha, beta) electrons of the input CI function

ap_id [int] Orbital index (0-based), for the creation operator

Returns: 2D array, row for alpha strings and column for beta strings. Note it has different number of columns to the input CI coefficients.

pyscf.fci.addons.**des_a**(*ci0*, *norb*, *neleca_nelecb*, *ap_id*)
Construct (N-1)-electron wavefunction by removing an alpha electron from the N-electron wavefunction.

... math:

```
|N-1\rangle = \hat{a}_p |N\rangle
```

Args:

ci0 [2D array] CI coefficients, row for alpha strings and column for beta strings.

norb [int] Number of orbitals.

(neleca,nelecb) [(int,int)] Number of (alpha, beta) electrons of the input CI function

ap_id [int] Orbital index (0-based), for the annihilation operator

Returns: 2D array, row for alpha strings and column for beta strings. Note it has different number of rows to the input CI coefficients

pyscf.fci.addons.**des_b**(*ci0*, *norb*, *neleca_nelecb*, *ap_id*)
Construct (N-1)-electron wavefunction by removing a beta electron from N-electron wavefunction.

Args:

ci0 [2D array] CI coefficients, row for alpha strings and column for beta strings.

norb [int] Number of orbitals.

(neleca,nelecb) [(int,int)] Number of (alpha, beta) electrons of the input CI function

ap_id [int] Orbital index (0-based), for the annihilation operator

Returns: 2D array, row for alpha strings and column for beta strings. Note it has different number of columns to the input CI coefficients.

pyscf.fci.addons.**det_overlap**(*string1*, *string2*, *norb*, *s=None*)
Determinants overlap on non-orthogonal one-particle basis

pyscf.fci.addons.**energy**(*h1e*, *eri*, *fcivec*, *norb*, *nelec*, *link_index=None*)
Compute the FCI electronic energy for given Hamiltonian and FCI vector.

pyscf.fci.addons.**fix_spin_**(*fciobj*, *shift=0.2*, *ss=None*, ***kwargs*)
If FCI solver cannot stick on spin eigenfunction, modify the solver by adding a shift on spin square operator

$$(H + shift * S^2)|\Psi\rangle = E|\Psi\rangle$$

Args: fciobj : An instance of `FCISolver`

Kwargs:

shift [float] Level shift for states which have different spin

ss [number] S^2 expection value == s*(s+1)

Returns A modified FCI object based on fciobj.

pyscf.fci.addons.**guess_wfnsym**(*ci*, *norb*, *nelec*, *orbsym*)
Guess the wavefunction symmetry based on the non-zero elements in the given CI coefficients.

Args:

ci [2D array] CI coefficients, row for alpha strings and column for beta strings.

norb [int] Number of orbitals.

nelec [int or 2-item list] Number of electrons, or 2-item list for (alpha, beta) electrons

orbsym [list of int] The irrep ID for each orbital.

Returns: Irrep ID

pyscf.fci.addons.**initguess_triplet**(*norb*, *nelec*, *binstring*)
Generate a triplet initial guess for FCI solver

pyscf.fci.addons.**large_ci**(*ci*, *norb*, *nelec*, *tol=0.1*, *return_strs=True*)
Search for the largest CI coefficients

pyscf.fci.addons.**overlap**(*bra*, *ket*, *norb*, *nelec*, *s=None*)
Overlap between two CI wavefunctions

Args:

s [2D array or a list of 2D array] The overlap matrix of non-orthogonal one-particle basis

pyscf.fci.addons.**reorder**(*ci*, *nelec*, *orbidxa*, *orbidxb=None*)
Reorder the CI coefficients, to adapt the reordered orbitals (The relation of the reordered orbitals and original orbitals is new = old[idx]). Eg.

The orbital ordering indices `orbidx = [2,0,1]` indicates the map old orbital a b c -> new orbital c a b. The strings are reordered as old-strings 0b011, 0b101, 0b110 == (1,2), (1,3), (2,3) <= apply orbidx to get orb-strings orb-strings (3,1), (3,2), (1,2) == 0B101, 0B110, 0B011 <= by gen_strings4orblist then argsort to translate the string representation to the address [2(=0B011), 0(=0B101), 1(=0B110)]

pyscf.fci.addons.**symm_initguess**(*norb*, *nelec*, *orbsym*, *wfnsym=0*, *irrep_nelec=None*)
Generate CI wavefunction initial guess which has the given symmetry.

Args:

norb [int] Number of orbitals.

nelec [int or 2-item list] Number of electrons, or 2-item list for (alpha, beta) electrons

orbsym [list of int] The irrep ID for each orbital.

Kwags:

wfnsym [int] The irrep ID of target symmetry

irrep_nelec [dict] Freeze occupancy for certain irreps

Returns: CI coefficients 2D array which has the target symmetry.

pyscf.fci.addons.**symmetrize_wfn**(*ci*, *norb*, *nelec*, *orbsym*, *wfnsym=0*)
Symmetrize the CI wavefunction by zeroing out the determinants which do not have the right symmetry.

Args:

ci [2D array] CI coefficients, row for alpha strings and column for beta strings.

norb [int] Number of orbitals.

nelec [int or 2-item list] Number of electrons, or 2-item list for (alpha, beta) electrons

orbsym [list of int] The irrep ID for each orbital.

Kwags:

wfnsym [int] The irrep ID of target symmetry

Returns: 2D array which is the symmetrized CI coefficients

pyscf.fci.addons.**transform_ci_for_orbital_rotation**(*ci*, *norb*, *nelec*, *u*)

Transform CI coefficients to the representation in new one-particle basis. Solving CI problem for Hamiltonian h1, h2 defined in old basis, CI_old = fci.kernel(h1, h2, ...) Given orbital rotation u, the CI problem can be either solved by transforming the Hamiltonian, or transforming the coefficients. CI_new = fci.kernel(u^T*h1*u, ...) = transform_ci_for_orbital_rotation(CI_old, u)

Args:

u [2D array or a list of 2D array] the orbital rotation to transform the old one-particle basis to new one-particle basis

1.10 symm – Point group symmetry and spin symmetry

This module offers the functions to detect point group symmetry, basis symmetriziation, Clebsch-Gordon coefficients. This module works as a plugin of PySCF package. Symmetry is not hard coded in each method.

PySCF supports D2h symmetry and linear molecule symmetry (Dooh and Coov). For D2h, the direct production of representations are

D2h	A1g	B1g	B2g	B3g	A1u	B1u	B2u	B3u
A1g	A1g							
B1g	B1g	A1g						
B2g	B2g	B3g	A1g					
B3g	B3g	B2g	B1g	A1g				
A1u	A1u	B1u	B2u	B3u	A1g			
B1u	B1u	A1u	B3u	B2u	B1g	A1g		
B2u	B2u	B3u	A1u	B1u	B2g	B3g	A1g	
B3u	B3u	B2u	B1u	A1u	B3g	B2g	B1g	A1g

The multiplication table for XOR operator is

XOR	000	001	010	011	100	101	110	111
000	000							
001	001	000						
010	010	011	000					
011	011	010	001	000				
100	100	101	110	111	000			
101	101	100	111	110	001	000		
110	110	111	100	101	010	011	000	
111	111	110	101	100	011	010	001	000

Comparing the two table, we notice that the two tables can be changed to each other with the mapping

D2h	XOR	ID
A1g	000	0
B1g	001	1
B2g	010	2
B3g	011	3
A1u	100	4
B1u	101	5
B2u	110	6
B3u	111	7

The XOR operator and the D2h subgroups have the similar relationships. We therefore use the XOR operator ID to assign the irreps (see `pyscf/symm/param.py`).

C2h	XOR	ID	C2v	XOR	ID	D2	XOR	ID
Ag	00	0	A1	00	0	A1	00	0
Bg	01	1	A2	01	1	B1	01	1
Au	10	2	B1	10	2	B2	10	2
Bu	11	3	B2	11	3	B3	11	3

Cs	XOR	ID	Ci	XOR	ID	C2	XOR	ID
A’	0	0	Ag	0	0	A	0	0
B”	1	1	Au	1	1	B	1	1

To easily get the relationship between the linear molecule symmetry and D2h/C2v, the ID for irreducible representations of linear molecule symmetry are chosen as (see `pyscf/symm/basis.py`)

$D_{\infty h}$	ID	D_{2h}	ID
A1g	0	Ag	0
A2g	1	B1g	1
A1u	5	B1u	5
A2u	4	Au	4
E1gx	2	B2g	2
E1gy	3	B3g	3
E1uy	6	B2u	6
E1ux	7	B3u	7
E2gx	10	Ag	0
E2gy	11	B1g	1
E2ux	15	B1u	5
E2uy	14	Au	4
E3gx	12	B2g	2
E3gy	13	B3g	3
E3uy	16	B2u	6
E3ux	17	B3u	7
E4gx	20	Ag	0
E4gy	21	B1g	1
E4ux	25	B1u	5
E4uy	24	Au	4
E5gx	22	B2g	2
E5gy	23	B3g	3
E5uy	26	B2u	6
E5ux	27	B3u	7

and

$C_{\infty v}$	ID	C_{2v}	ID
A1	0	A1	0
A2	1	A2	1
E1x	2	B1	2
E1y	3	B2	3
E2x	10	A1	0
E2y	11	A2	1
E3x	12	B1	2
E3y	13	B2	3
E4x	20	A1	0
E4y	21	A2	1
E5x	22	B1	2
E5y	23	B2	3

So that, the subduction from linear molecule symmetry to D2h/C2v can be achieved by the modular operation `%10`.

In many output messages, the irreducible representations are labeld with the IDs instead of the irreps' symbols. We can use `symm.addons.irrep_id2name()` to convert the ID to irreps' symbol, e.g.:

```
>>> from pyscf import symm
>>> [symm.irrep_id2name('Dooh', x) for x in [7, 6, 0, 10, 11, 0, 5, 3, 2, 5, 15, 14]]
['E1ux', 'E1uy', 'A1g', 'E2gx', 'E2gy', 'A1g', 'A1u', 'E1gy', 'E1gx', 'A1u', 'E2ux',
 ↪'E2uy']
```

1.10.1 Enabling symmetry in other module

- SCF

  To control the HF determinant symmetry, one can assign occupancy for particular irreps, e.g.

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

from pyscf import gto
from pyscf import scf

'''
Specify irrep_nelec to control the wave function symmetry
'''


mol = gto.Mole()
mol.build(
    verbose = 0,
    atom = '''
        C     0.   0.    0.625
        C     0.   0.  -0.625 ''',
    basis = 'cc-pVDZ',
    spin = 0,
    symmetry = True,
)

mf = scf.RHF(mol)

# Frozen occupancy
```

```
# 'A1g': 4 electrons
# 'E1gx': 2 electrons
# 'E1gy': 2 electrons
# Rest 4 electrons are put in irreps A1u, E1ux, E1uy ... based on Aufbau principle
mf.irrep_nelec = {'A1g': 4, 'E1gx': 2, 'E1gy': 2}
e = mf.kernel()
print('E = %.15g  ref = -74.1112374269129' % e)


mol.symmetry = 'D2h'
mol.charge = 1
mol.spin = 1
mol.build(dump_input=False, parse_arg=False)
mf = scf.RHF(mol)

# Frozen occupancy
# 'Ag': 2 alpha, 1 beta electrons
# 'B1u': 4 electrons
# 'B2u': 2 electrons
# 'B3u': 2 electrons
mf.irrep_nelec = {'Ag': (2,1), 'B1u': 4, 'B2u': 2, 'B3u': 2,}
e = mf.kernel()
print('E = %.15g  ref = -74.4026583773135' % e)

# Frozen occupancy
# 'Ag': 4 electrons
# 'B1u': 2 alpha, 1 beta electrons
# 'B2u': 2 electrons
# 'B3u': 2 electrons
mf.irrep_nelec = {'Ag': 4, 'B1u': (2,1), 'B2u': 2, 'B3u': 2,}
e = mf.kernel()
print('E = %.15g  ref = -74.8971476600812' % e)
```

- FCI

  FCI wavefunction symmetry can be controlled by initial guess. Function `fci.addons.symm_initguess()` can generate the FCI initial guess with the right symmetry.

- MCSCF

  The symmetry of active space in the CASCI/CASSCF calculations can controlled

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

'''
Active space can be adjusted by specifing the number of orbitals for each irrep.
'''

from pyscf import gto, scf, mcscf

mol = gto.Mole()
mol.build(
    atom = 'N  0  0  0; N  0  0  2',
    basis = 'ccpvtz',
    symmetry = True,
)
```

```
myhf = scf.RHF(mol)
myhf.kernel()

mymc = mcscf.CASSCF(myhf, 8, 4)
mo = mcscf.sort_mo_by_irrep(mymc, myhf.mo_coeff,
                            {'E1gx':2, 'E1gy':2, 'E1ux':2, 'E1uy':2})
mymc.kernel(mo)
```

- MP2 and CCSD

  Point group symmetry are not supported in CCSD, MP2.

Program reference

1.10.2 geom

pyscf.symm.geom.**alias_axes**(*axes*, *ref*)
Rename axes, make it as close as possible to the ref axes

pyscf.symm.geom.**detect_symm**(*atoms*, *basis=None*, *verbose=2*)
Detect the point group symmetry for given molecule.

Return group name, charge center, and nex_axis (three rows for x,y,z)

pyscf.symm.geom.**rotation_mat**(*vec*, *theta*)
rotate angle theta along vec new(x,y,z) = R * old(x,y,z)

pyscf.symm.geom.**symm_identical_atoms**(*gpname*, *atoms*)
Requires

1.10.3 basis

Generate symmetry adapted basis

pyscf.symm.basis.**linearmole_symm_descent**(*gpname*, *irrepid*)
Map irreps to D2h or C2v

1.10.4 addons

pyscf.symm.addons.**eigh**(*h*, *orbsym*)
Solve eigenvalue problem based on the symmetry information for basis. See also pyscf/lib/linalg_helper.py `eigh_by_blocks()`

Examples:

```
>>> from pyscf import gto, symm
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz', symmetry=True)
>>> c = numpy.hstack(mol.symm_orb)
>>> vnuc_so = reduce(numpy.dot, (c.T, mol.intor('int1e_nuc_sph'), c))
>>> orbsym = symm.label_orb_symm(mol, mol.irrep_name, mol.symm_orb, c)
>>> symm.eigh(vnuc_so, orbsym)
(array([-4.50766885, -1.80666351, -1.7808565 , -1.7808565 , -1.74189134,
        -0.98998583, -0.98998583, -0.40322226, -0.30242374, -0.07608981]),
 ...)
```

pyscf.symm.addons.**irrep_id2name**(*gpname*, *irrep_id*)
Convert the internal irrep ID to irrep symbol

Args:

gpname [str] The point group symbol

irrep_id [int] See IRREP_ID_TABLE in pyscf/symm/param.py

Returns: Irrep sybmol, str

pyscf.symm.addons.**irrep_name2id**(*gpname*, *symb*)
Convert the irrep symbol to internal irrep ID

Args:

gpname [str] The point group symbol

symb [str] Irrep symbol

Returns: Irrep ID, int

pyscf.symm.addons.**label_orb_symm**(*mol*, *irrep_name*, *symm_orb*, *mo*, *s=None*, *check=True*, *tol=1e-09*)
Label the symmetry of given orbitals

irrep_name can be either the symbol or the ID of the irreducible representation. If the ID is provided, it returns the numeric code associated with XOR operator, see `symm.param.IRREP_ID_TABLE()`

Args: mol : an instance of `Mole`

irrep_name [list of str or int] A list of irrep ID or name, it can be either mol.irrep_id or mol.irrep_name. It can affect the return "label".

symm_orb [list of 2d array] the symmetry adapted basis

mo [2d array] the orbitals to label

Returns: list of symbols or integers to represent the irreps for the given orbitals

Examples:

```
>>> from pyscf import gto, scf, symm
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz',verbose=0, symmetry=1)
>>> mf = scf.RHF(mol)
>>> mf.kernel()
>>> symm.label_orb_symm(mol, mol.irrep_name, mol.symm_orb, mf.mo_coeff)
['Ag', 'B1u', 'Ag', 'B1u', 'B2u', 'B3u', 'Ag', 'B2g', 'B3g', 'B1u']
>>> symm.label_orb_symm(mol, mol.irrep_id, mol.symm_orb, mf.mo_coeff)
[0, 5, 0, 5, 6, 7, 0, 2, 3, 5]
```

pyscf.symm.addons.**route**(*target*, *nelec*, *orbsym*)
Pick orbitals to form a determinant which has the right symmetry. If solution is not found, return []

pyscf.symm.addons.**std_symb**(*gpname*)
std_symb('d2h') returns D2h; std_symb('D2H') returns D2h

pyscf.symm.addons.**symmetrize_orb**(*mol*, *mo*, *orbsym=None*, *s=None*, *check=False*)
Symmetrize the given orbitals.

This function is different to the *symmetrize_space()*: In this function, each orbital is symmetrized by removing non-symmetric components. *symmetrize_space()* symmetrizes the entire space by mixing different orbitals.

Note this function might return non-orthorgonal orbitals. Call *symmetrize_space()* to find the symmetrized orbitals that are close to the given orbitals.

Args:

mo [2D float array] The orbital space to symmetrize

Kwargs:

orbsym [integer list] Irrep id for each orbital. If not given, the irreps are guessed by calling *label_orb_symm()*.

s [2D float array] Overlap matrix. If given, use this overlap than the the overlap of the input mol.

Returns: 2D orbital coefficients

Examples:

```
>>> from pyscf import gto, symm, scf
>>> mol = gto.M(atom = 'C  0  0  0; H  1  1  1; H -1 -1  1; H  1 -1 -1; H -1  1 -1
 ↪',
...             basis = 'sto3g')
>>> mf = scf.RHF(mol).run()
>>> mol.build(0, 0, symmetry='D2')
>>> mo = symm.symmetrize_orb(mol, mf.mo_coeff)
>>> print(symm.label_orb_symm(mol, mol.irrep_name, mol.symm_orb, mo))
['A', 'A', 'B1', 'B2', 'B3', 'A', 'B1', 'B2', 'B3']
```

pyscf.symm.addons.**symmetrize_space**(*mol*, *mo*, *s=None*, *check=True*)

Symmetrize the given orbital space.

This function is different to the *symmetrize_orb()*: In this function, the given orbitals are mixed to reveal the symmtery; *symmetrize_orb()* projects out non-symmetric components for each orbital.

Args:

mo [2D float array] The orbital space to symmetrize

Kwargs:

s [2D float array] Overlap matrix. If not given, overlap is computed with the input mol.

Returns: 2D orbital coefficients

Examples:

```
>>> from pyscf import gto, symm, scf
>>> mol = gto.M(atom = 'C  0  0  0; H  1  1  1; H -1 -1  1; H  1 -1 -1; H -1  1 -1
 ↪',
...             basis = 'sto3g')
>>> mf = scf.RHF(mol).run()
>>> mol.build(0, 0, symmetry='D2')
>>> mo = symm.symmetrize_space(mol, mf.mo_coeff)
>>> print(symm.label_orb_symm(mol, mol.irrep_name, mol.symm_orb, mo))
['A', 'A', 'A', 'B1', 'B1', 'B2', 'B2', 'B3', 'B3']
```

1.10.5 Clebsch Gordon coefficients

1.11 df — Density fitting

1.11.1 Introduction

The `df` module provides the fundamental functions to handle the 3-index tensors required by the density fitting (DF) method or the resolution of identity (RI) approximation. Specifically, it includes the functions to compute the 3-center 2-electron AO integrals, the DF/RI 3-index tensor in the form of Cholesky decomposed integral tensor ($(ij|kl) = V_{ij,x}V_{kl,x}$), the AO to MO integral transformation of the 3-index tensor, as well as the functions to generate the density fitting basis.

The `density_fit()` method can utilize the DF method at SCF and MCSCF level:

```
from pyscf import gto, scf, mcscf
mol = gto.M(atom='N 0 0 0; N 0 0 1.2', basis='def2-tzvp')
mf = scf.RHF(mol).density_fit().run()
mc = mcscf.CASSCF(mf, 8, 10).density_fit().run()
```

Once the DF method is enabled at the SCF level, all the post-SCF methods will automatically enable the DF method, for example:

```
from pyscf import gto, dft, tddft
mol = gto.M(atom='N 0 0 0; N 0 0 1.2', basis='def2-tzvp')
mf = dft.RKS(mol).density_fit().run()
td = tddft.TDA(mf).run()
print(td.e)
```

In PySCF, DF is implemented at different level of implementations for different methods. They are summarized in the following table

Methods	Fake-ERI	Native DF	Properties with DF
HF/DFT	Yes	Yes	
Generlized HF/DFT	Yes	No	
Relativistic HF	Yes	Yes	
TDDFT	Yes	Yes	
RCCSD	Yes	Yes	
UCCSD	Yes	No	
RCCSD(T)	Yes	No	
EOM-CCSD	Yes	No	
RMP2	Yes	No	
UMP2	Yes	No	
PBC HF/DFT	Yes	Yes	
PBC TDDFT	Yes	Yes	
PBC Gamma-point CCSD	Yes	Yes	
PBC k-points CCSD	Yes	No	

Fake-ERI means to mimic the 4-center 2-electron repulsion integrals (ERI) by precontracting the DF 3-index tensor. This is the simplest way to enable DF integrals, although the fake-ERI mechanism may require huge amount of memory also may be slow in performance. It provides the most convenient way to embed the DF integrals in the existing code, thus it is supported by almost every method in PySCF. It is particularly important in the periodic code. Using the fake-ERIs allows us to call all quantum chemistry methods developed at molecular level in the Γ-point calculations without modifying any existing molecular code. See also the *pbc.df — PBC denisty fitting* module.

Some methods have native DF implementation. This means the performance of the DF technique has been considered

in the code. In these methods, DF approximation generally runs faster than the regular scheme without integral approximation and also consumes less memory or disk space.

When density fitting is enabled in a method, a `with_df` object will be generated and attached to the method object. `with_df` is the object to hold all DF relevant quantiles, such as the DF basis, the file to save the 3-index tensor, the amount of memory to use etc. You can modify the attributes of `with_df` to get more control over the DF methods. In the SCF and MCSCF methods, setting `with_df` to `None` will switch off the DF approximation. In the periodic code, all two-electron integrals are evaluated by DF approximations. There are four different types of DF schemes (`FFTDF`, `AFTDF`, `GDF`, `MDF` see *pbc.df — PBC denisty fitting*), available in the periodic code. By assigning different DF object to `with_df`, different DF schemes can be applied in the PBC calculations.

DF auxiliary basis

The default auxiliary basis set are generated by function `pyscf.df.addons.make_basis()` based on the orbital basis specified in the calculation according to the rules defined in `pyscf.df.addons.DEFAULT_AUXBASIS`. Specifically, the *jkfit* basis in the first column is used for Hartree-Fock or DFT methods, and the *ri* basis in the second column is used for correlation calculations. These optimized auxiliary basis sets are obtained from http://www.psicode.org/psi4manual/master/basissets_byfamily.html If optimized auxiliary basis set was not found for the orbital basis set, even-tempered Gaussian functions are generated automatically.

Specifying auxiliary basis is a common requirement in the real applications. For example, the default auxiliary basis set for the pure DFT calculations may be over complete since it is designed to represent both the Coulomb and HF exchange matrix. Coulomb fitting basis such as Weigend-cfit basis or Ahlrichs-cfit basis are often enough to obtain chemical accuracy. To control the fitting basis in DF method, You can change the value of `with_df.auxbasis` attribute. The input format of auxiliary fitting basis is exactly the same to the input format of orbital *basis* set. For example:

```
from pyscf import gto, dft
mol = gto.M(atom='N 0 0 0; N 0 0 1.2', basis='def2-tzvp')
mf = dft.RKS(mol)
mf.xc = 'pbe,pbe'
mf.run()  # -109.432313679876
mf = mf.density_fit()
mf.run()  # -109.432329411505
mf.with_df.auxbasis = 'weigend'
mf.run()  # -109.432334646584
```

More examples for inputing auxiliary basis in the DF calculation can be found in `examples/df/01-auxbasis.py`.

Even-tempered auxiliary Gaussian basis

The even-tempered auxiliary Gaussian basis is generated by function `aug_etb()`:

```
from pyscf import gto, df
mol = gto.M(atom='N 0 0 0; N 0 0 1.2', basis='ccpvdz')
print(mol.nao_nr())                              # 28
auxbasis = df.aug_etb(mol)
print(df.make_auxmol(mol, auxbasis).nao_nr())  # 200
auxbasis = df.aug_etb(mol, beta=1.6)
print(df.make_auxmol(mol, auxbasis).nao_nr())  # 338
```

Here the `make_auxmol()` function converts the auxbasis to a `Mole` object which can be used to evaluate the analytical integrals the same way as the regular `Mole` object. The formula to generate the exponents ζ of the even-

tempered auxiliary basis are

$$\varphi = r^l \exp(-\zeta_{il} r^2), \quad i = 0..n$$
$$\zeta_{il} = \alpha \times \beta^i : label : etb$$

The default value of β is 2.3. α and the number of auxiliary basis n is determined based on the orbital basis. Given the orbital basis

$$\chi = r^l \exp(-\alpha_l r^2)$$

the orbital pair on the same center produces a new one-center basis

$$\chi\chi' = r^{l+l'} \exp(-(\alpha_l + \alpha'_l) r^2) = r^L \exp(-\alpha_L r^2)$$

The minimal α_L in all orbital pairs is assigned to α in (**??**). Then n is estimated to make the largest auxiliary exponent ζ as close as possible to the maximum α_L. The size of generated even-tempered Gaussian basis is typically 5 - 10 times of the size of the orbital basis, or 2 - 3 times more than the optimized auxiliary basis. (Note the accuracy of this even-tempered auxiliary basis is not fully benchmarked. The error is close to the optimized auxiliary basis in our tests.)

Saving/Loading DF integral tensor

Although it is not expensive to compute DF integral tensor in the molecular calculation, saving/loading the 3-index tensor is still useful since it provides an alternative way, different to the attribute `_eri` of mean-field object (see *Customizing Hamiltonian*), to customize the Hamiltonian.

In the DF-SCF method, the 3-index tensor is held in the `with_df` object. The `with_df` object (see *pyscf.df.df.DF* class) provides two attributes `_cderi_to_save` and `_cderi` to access the DF 3-index integrals.

If a DF integral tensor is assigned to `_cderi`, the integrals will be used in the DF calculation. The DF integral tensor can be either a numpy array or an HDF5 file on disk. When the DF integrals are provided in the HDF5 file, the integral needs to be stored under the dataset `'j3c'`:

```
import numpy
import h5py
from pyscf import gto, scf, df
mol = gto.M(atom='H 0 0 0; H 1 0 1; H 0 1 1; H 1 1 0', basis='sto3g')
nao = mol.nao_nr()
with h5py.File('df_ints.h5', 'w') as f:
    f['j3c'] = numpy.random.random((10,nao*(nao+1)//2))
mf = scf.RHF(mol).density_fit()
mf.with_df._cderi = 'df_ints.h5'
mf.kernel()
```

As shown in the above example, the integral tensor $V_{x,ij}$ provided in `_cderi` should be a 2D array in C (row-major) convention. Its first index corresponds to the auxiliary basis and the second combined index `ij` is the orbital pair index. When load DF integrals, we assumed hermitian symmetry between the two orbital index, ie only the elements $i \geq j$ are left in the DF integral tensor. Thus the DF integral tensor should be a 2D array, with shape `(M,N*(N+1)/2)`, where `M` is the number of auxiliary functions, N is the number of orbitals.

If `_cderi` is not specified, the DF integral tensor will be generated during the calculation and stored to the file that the attribute `_cderi_to_save` points to. By default, it is a random file and the random file will be deleted if the calculation finishes successfully. You can find the filename in the output log (when `with.verbose > 3`, for example:

```
******** <class 'pyscf.df.df.DF'> flags ********
auxbasis = None
max_memory = 20000
_cderi_to_save = /scratch/tmp6rGrSD
```

If the calculation is terminated problematically with error or any other reasons, you can reuse the DF integrals in the next calculation by assigning the integral file to `_cderi`. Overwriting `_cderi_to_save` with a filename will make the program save the DF integrals in the given filename regardless whether the calculation is succeed or failed. See also the example `pyscf/examples/df/10-access_df_integrals.py`.

Precomputing the DF integral tensor

The DF integral tensor can be computed without initialization the `with_df` object. Functions `cholesky_eri()` defined in `df.incore` and `df.outcore` can generate DF integral tensor in memory or in a HDF5 file:

```
from pyscf import gto, df
mol = gto.M(atom='N 0 0 0; N 1 1 1', basis='ccpvdz')
cderi = df.incore.cholesky_eri(mol, auxbasis='ccpvdz-jkfit')
df.outcore.cholesky_eri(mol, 'df_ints.h5', auxbasis='ccpvdz-jkfit')
```

These `cderi` integrals has the same data structure as the one generated in `with_df` object. They can be directly used in the DF type calculations:

```
from pyscf import scf
mf = scf.RHF(mol).density_fit()
mf.with_df._cderi = cderi
mf.kernel()

mf.with_df._cderi = 'df_ints.h5'
mf.kernel()
```

Approximating orbital hessian in SCF and MCSCF

Orbital hessian is required by the second order SCF solver or MCSCF solver. In many systems, approximating the orbital hessian has negligible effects to the convergence and the solutions of the SCF or MCSCF orbital optimization procedure. Using DF method to approximate the orbital hessian can improve the overall performance. For example, the following code enables the DF approximation to the orbital hessian in SCF calculation:

```
from pyscf import gto, scf
mol = gto.M(atom='N 0 0 0; O 0 0 1.5', spin=1, basis='ccpvdz')
mf = scf.RHF(mol).newton().density_fit().run(verbose=4)  # converged SCF energy = -
↪129.0896469563
mf = scf.RHF(mol).run(verbose=4)                         # converged SCF energy = -
↪129.0896469563
```

The approximation to orbital hessian does not change the SCF result. In the above example, it produces the same solution to the regular SCF result. Similarly, when the DF approximation is used with CASSCF orbital hessian, the CASSCF result should not change. Continuing the above example, we can use the `mcscf.approx_hessian()` function to change the orbital hessian of the given CASSCF object:

```
from pyscf import mcscf
mc = mcscf.approx_hessian(mcscf.CASSCF(mf, 8, 11)).run()  # -129.283077136
mc = mcscf.CASSCF(mf, 8, 11).run()                        # -129.283077136
```

Note: In the second order SCF solver, the order to apply the density_fit and newton methods affects the character of the resultant SCF object. For example, the statement `mf = scf.RHF(mol).density_fit().newton()` first produces a DFHF object then enable the second order Newton solver for the DFHF object. The resultant SCF object is a DFHF object. See more examples in `examples/scf/23-decorate_scf.py`

1.11.2 Program reference

DF class

class pyscf.df.df.**DF**(*mol*)
Object to hold 3-index tensor

Attributes:

auxbasis [str or dict] Same input format as `Mole.basis`

auxmol [Mole object] Read only Mole object to hold the auxiliary basis. auxmol is generated automatically in the initialization step based on the given auxbasis. It is used in the rest part of the code to determine the problem size, the integral batches etc. This object should NOT be modified.

_cderi_to_save [str] If _cderi_to_save is specified, the DF integral tensor will be saved in this file.

_cderi [str or numpy array] If _cderi is specified, the DF integral tensor will be read from this HDF5 file (or numpy array). When the DF integral tensor is provided from the HDF5 file, it has to be stored under the dataset 'j3c'. The DF integral tensor $V_{x,ij}$ should be a 2D array in C (row-major) convention, where x corresponds to index of auxiliary basis, and the combined index ij is the orbital pair index. The hermitian symmetry is assumed for the combined ij index, ie the elements of $V_{x,i,j}$ with $i \geq j$ are existed in the DF integral tensor. Thus the shape of DF integral tensor is (M,N*(N+1)/2), where M is the number of auxbasis functions and N is the number of basis functions of the orbital basis.

blockdim [int] When reading DF integrals from disk the chunk size to load. It is used to improve the IO performance.

df.incore

pyscf.df.incore.**aux_e1**(*mol*, *auxmol*, *intor='int3c2e_sph'*, *aosym='s1'*, *comp=1*, *out=None*)
3-center 2-electron AO integrals (Llij), where L is the auxiliary basis.

pyscf.df.incore.**aux_e2**(*mol*, *auxmol*, *intor='int3c2e_sph'*, *aosym='s1'*, *comp=1*, *out=None*)
3-center AO integrals (ij|L), where L is the auxiliary basis.

pyscf.df.incore.**cholesky_eri**(*mol*, *auxbasis='weigend+etb'*, *auxmol=None*, *int3c='int3c2e_sph'*, *aosym='s2ij'*, *int2c='int2c2e_sph'*, *comp=1*, *verbose=0*, *fauxe2=<function aux_e2>*)

Returns: 2D array of (naux,nao*(nao+1)/2) in C-contiguous

pyscf.df.incore.**fill_2c2e**(*mol*, *auxmol*, *intor='int2c2e_sph'*, *comp=1*, *hermi=1*, *out=None*)
2-center 2-electron AO integrals for auxiliary basis (auxmol)

pyscf.df.incore.**format_aux_basis**(*mol*, *auxbasis='weigend+etb'*)
See also pyscf.df.addons.make_auxmol.

This funciton is defined for backward compatibility.

df.outcore

pyscf.df.outcore.**cholesky_eri**(*mol*, *erifile*, *auxbasis='weigend+etb'*, *dataname='j3c'*, *tmpdir=None*, *int3c='int3c2e_sph'*, *aosym='s2ij'*, *int2c='int2c2e_sph'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *auxmol=None*, *verbose=0*)

3-center 2-electron AO integrals

pyscf.df.outcore.**cholesky_eri_b**(*mol*, *erifile*, *auxbasis='weigend+etb'*, *dataname='j3c'*, *int3c='int3c2e_sph'*, *aosym='s2ij'*, *int2c='int2c2e_sph'*, *comp=1*, *ioblk_size=256*, *auxmol=None*, *verbose=3*)

3-center 2-electron AO integrals

pyscf.df.outcore.**general**(*mol*, *mo_coeffs*, *erifile*, *auxbasis='weigend+etb'*, *dataname='eri_mo'*, *tmpdir=None*, *int3c='int3c2e_sph'*, *aosym='s2ij'*, *int2c='int2c2e_sph'*, *comp=1*, *max_memory=2000*, *ioblk_size=256*, *verbose=0*, *compact=True*)

Transform ij of (ij|L) to MOs.

df.addons

pyscf.df.addons.**aug_etb**(*mol*, *beta=2.3*)

To generate the even-tempered auxiliary Gaussian basis

pyscf.df.addons.**aug_etb_for_dfbasis**(*mol*, *dfbasis='weigend'*, *beta=2.3*, *start_at='Rb'*)

augment weigend basis with even-tempered gaussian basis exps = alpha*beta^i for i = 1..N

class pyscf.df.addons.**load**(*eri*, *dataname='j3c'*)

load 3c2e integrals from hdf5 file. It can be used in the context manager:

with load(cderifile) as eri: print eri.shape

pyscf.df.addons.**make_auxbasis**(*mol*, *mp2fit=False*)

Depending on the orbital basis, generating even-tempered Gaussians or the optimized auxiliary basis defined in DEFAULT_AUXBASIS

pyscf.df.addons.**make_auxmol**(*mol*, *auxbasis=None*)

Generate a fake Mole object which uses the density fitting auxbasis as the basis sets. If auxbasis is not specified, the optimized auxiliary fitting basis set will be generated according to the rules recorded in pyscf.df.addons.DEFAULT_AUXBASIS. If the optimized auxiliary basis is not available (either not specified in DEFAULT_AUXBASIS or the basis set of the required elements not defined in the optimized auxiliary basis), even-tempered Gaussian basis set will be generated.

See also the paper JCTC, 13, 554 about the generation of auxiliary fitting basis.

1.12 dft — Density functional theory

1.12.1 Customizing XC functional

XC functional of DFT methods can be customized. The simplest way to customize XC functional is to assigned a string expression to `mf.xc`:

```
from pyscf import gto, dft
mol = gto.M(atom='H  0  0  0; F  0.9  0  0', basis='6-31g')
mf = dft.RKS(mol)
mf.xc = 'HF*0.2 + .08*LDA + .72*B88, .81*LYP + .19*VWN'
```

```
mf.kernel()
mf.xc = 'HF*0.5 + .08*LDA + .42*B88, .81*LYP + .19*VWN'
mf.kernel()
mf.xc = 'HF*0.8 + .08*LDA + .12*B88, .81*LYP + .19*VWN'
mf.kernel()
mf.xc = 'HF'
mf.kernel()
```

The XC functional string is parsed against the following rules.

- The given functional description must be a one-line string.
- The functional description is case-insensitive.
- The functional description string has two parts, separated by `,`. The first part describes the exchange functional, the second is the correlation functional. - If `,` was not appeared in string, the entire string is considered as

  X functional.

  - To neglect X functional (just apply C functional), leave blank in the first part, eg `mf.xc=',vwn'` for pure VWN functional
- The functional name can be placed in arbitrary order. Two names needs to be separated by operators `+` or `-`. Blank spaces are ignored. NOTE the parser only reads operators `+ -*.` `/` is not supported.
- A functional name is associated with one factor. If the factor is not given, it is assumed equaling 1.
- String `'HF'` stands for exact exchange (HF K matrix). It is allowed to put `'HF'` in C (correlation) functional part.
- Be careful with the libxc convention on GGA functional, in which the LDA contribution is included.

Another way to customize XC functional is to redefine the `eval_xc()` method of numerical integral class:

```
mol = gto.M(atom='H 0 0 0; F 0.9 0 0', basis = '6-31g')
mf = dft.RKS(mol)
def eval_xc(xc_code, rho, spin=0, relativity=0, deriv=1, verbose=None):
    # A fictitious XC functional to demonstrate the usage
    rho0, dx, dy, dz = rho
    gamma = (dx**2 + dy**2 + dz**2)
    exc = .01 * rho0**2 + .02 * (gamma+.001)**.5
    vrho = .01 * 2 * rho0
    vgamma = .02 * .5 * (gamma+.001)**(-.5)
    vlapl = None
    vtau = None
    vxc = (vrho, vgamma, vlapl, vtau)
    fxc = None  # 2nd order functional derivative
    kxc = None  # 3rd order functional derivative
    return exc, vxc, fxc, kxc
dft.libxc.define_xc_(mf._numint, eval_xc, xctype='GGA')
mf.kernel()
```

By calling `dft.libxc.define_xc_()` function, the customized `eval_xc()` function is patched to the numerical integration class `mf._numint` dynamically.

More examples of customizing DFT XC functional can be found in `examples/dft/24-custom_xc_functional.py` and `examples/dft/24-define_xc_functional.py`.

1.12.2 Program reference

Non-relativistic restricted Kohn-Sham

class pyscf.dft.rks.**RKS**(*mol*)

Restricted Kohn-Sham SCF base class. non-relativistic RHF.

Attributes:

verbose [int] Print level. Default value equals to Mole.verbose

max_memory [float or int] Allowed memory in MB. Default equals to Mole.max_memory

chkfile [str] checkpoint file to save MOs, orbital energies etc.

conv_tol [float] converge threshold. Default is 1e-10

conv_tol_grad [float] gradients converge threshold. Default is sqrt(conv_tol)

max_cycle [int] max number of iterations. Default is 50

init_guess [str] initial guess method. It can be one of 'minao', 'atom', '1e', 'chkfile'. Default is 'minao'

diis [boolean or object of DIIS class listed in scf.diis] Default is diis.SCF_DIIS. Set it to None to turn off DIIS.

diis_space [int] DIIS space size. By default, 8 Fock matrices and errors vector are stored.

diis_start_cycle [int] The step to start DIIS. Default is 1.

diis_file: 'str' File to store DIIS vectors and error vectors.

level_shift [float or int] Level shift (in AU) for virtual space. Default is 0.

direct_scf [bool] Direct SCF is used by default.

direct_scf_tol [float] Direct SCF cutoff threshold. Default is 1e-13.

callback [function(envs_dict) => None] callback function takes one dict as the argument which is generated by the builtin function locals(), so that the callback function can access all local variables in the current envrionment.

conv_check [bool] An extra cycle to check convergence after SCF iterations.

Saved results

converged [bool] SCF converged or not

e_tot [float] Total HF energy (electronic energy plus nuclear repulsion)

mo_energy : Orbital energies

mo_occ Orbital occupancy

mo_coeff Orbital coefficients

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1', basis='cc-pvdz')
>>> mf = scf.hf.SCF(mol)
>>> mf.verbose = 0
>>> mf.level_shift = .4
>>> mf.scf()
-1.0811707843775884
```

Attributes for RKS:

xc [str] 'X_name,C_name' for the XC functional. Default is 'lda,vwn'

grids [Grids object] grids.level (0 - 9) big number for large mesh grids. Default is 3

radii_adjust

radi.treutler_atomic_radii_adjust (default)
radi.becke_atomic_radii_adjust
None : to switch off atomic radii adjustment

grids.atomic_radii

radi.BRAGG_RADII (default)
radi.COVALENT_RADII
None : to switch off atomic radii adjustment

grids.radi_method scheme for radial grids

radi.treutler (default)
radi.delley
radi.mura_knowles
radi.gauss_chebyshev

grids.becke_scheme weight partition function

gen_grid.original_becke (default)
gen_grid.stratmann

grids.prune scheme to reduce number of grids

gen_grid.nwchem_prune (default)
gen_grid.sg1_prune
gen_grid.treutler_prune
None : to switch off grids pruning

grids.symmetry True/False to symmetrize mesh grids (TODO)

grids.atom_grid 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.

small_rho_cutoff [float] Drop grids if their contribution to total electrons smaller than this cutoff value. Default is 1e-7.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz',␣
↪verbose=0)
>>> mf = dft.RKS(mol)
>>> mf.xc = 'b3lyp'
>>> mf.kernel()
-76.415443079840458
```

energy_elec(*ks*, *dm*, *h1e=None*, *vhf=None*)
Electronic part of RKS energy.

Args: ks : an instance of DFT class

dm [2D ndarray] one-partical density matrix

h1e [2D ndarray] Core hamiltonian

Returns: RKS electronic energy and the 2-electron part contribution

get_veff(*ks*, *mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Coulomb + XC functional

Note: This function will change the ks object.

Args:

ks [an instance of *RKS*] XC functional are controlled by ks.xc attribute. Attribute ks.grids might be initialized.

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. If not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference Vxc potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

Returns: matrix Veff = J + Vxc. Veff can be a list matrices, if the input dm is a list of density matrices.

pyscf.dft.rks.**energy_elec**(*ks*, *dm*, *h1e=None*, *vhf=None*)
Electronic part of RKS energy.

Args: ks : an instance of DFT class

dm [2D ndarray] one-partical density matrix

h1e [2D ndarray] Core hamiltonian

Returns: RKS electronic energy and the 2-electron part contribution

pyscf.dft.rks.**get_veff**(*ks*, *mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Coulomb + XC functional

Note: This function will change the ks object.

Args:

ks [an instance of *RKS*] XC functional are controlled by ks.xc attribute. Attribute ks.grids might be initialized.

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

dm_last [ndarray or a list of ndarrays or 0] The density matrix baseline. If not 0, this function computes the increment of HF potential w.r.t. the reference HF potential matrix.

vhf_last [ndarray or a list of ndarrays or 0] The reference Vxc potential matrix.

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

Returns: matrix Veff = J + Vxc. Veff can be a list matrices, if the input dm is a list of density matrices.

Non-relativistic Unrestricted Kohn-Sham

class pyscf.dft.uks.**UKS**(*mol*)
Unrestricted Kohn-Sham See pyscf/dft/rks.py RKS class for the usage of the attributes

get_veff(*ks*, *mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Coulomb + XC functional for UKS. See pyscf/dft/rks.py *get_veff()* fore more details.

pyscf.dft.uks.**get_veff**(*ks*, *mol=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*)
Coulomb + XC functional for UKS. See pyscf/dft/rks.py *get_veff()* fore more details.

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)

class pyscf.dft.gen_grid.**Grids**(*mol*)
DFT mesh grids

Attributes for Grids:

level [int (0 - 9)] big number for large mesh grids, default is 3

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.

level [int] To control the number of radial and angular grids. The default level 3 corresponds to (50,302) for H, He; (75,302) for second row; (80~105,434) for rest.

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()
```

gen_atomic_grids (*mol*, *atom_grid=None*, *radi_method=None*, *level=None*, *prune=None*)
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.

gen_partition (*mol*, *atom_grids_tab*, *radii_adjust=None*, *atomic_radii=array([0., 0.66140414, 2.64561657, 2.74010288, 1.98421243, 1.60626721, 1.32280829, 1.22832198, 1.13383567, 0.94486306, 2.83458919, 3.40150702, 2.83458919, 2.36215766, 2.07869874, 1.88972612, 1.88972612, 1.88972612, 3.40150702, 4.15739747, 3.40150702, 3.0235618, 2.64561657, 2.55113027, 2.64561657, 2.64561657, 2.64561657, 2.55113027, 2.55113027, 2.55113027, 2.55113027, 2.45664396, 2.36215766, 2.17318504, 2.17318504, 2.17318504, 3.59047964, 4.44085639, 3.77945225, 3.40150702, 2.92907549, 2.74010288, 2.74010288, 2.55113027, 2.45664396, 2.55113027, 2.64561657, 3.0235618, 2.92907549, 2.92907549, 2.74010288, 2.74010288, 2.64561657, 2.64561657, 3.96842486, 4.91328792, 4.06291117, 3.68496594, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.40150702, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 2.92907549, 2.74010288, 2.55113027, 2.55113027, 2.45664396, 2.55113027, 2.55113027, 2.55113027, 2.83458919, 3.59047964, 3.40150702, 3.0235618, 3.59047964, 2.74010288, 3.96842486, 3.40150702, 4.06291117, 3.68496594, 3.40150702, 3.40150702, 3.30702072])*, *becke_scheme=<function original_becke>*)
Generate the mesh grid coordinates and weights for DFT numerical integration. We can change radii_adjust, becke_scheme functions to generate different meshgrid.

Returns: grid_coord and grid_weight arrays. grid_coord array has shape (N,3); weight 1D array has N elements.

make_mask (*mol=None*, *coords=None*, *relativity=0*, *shls_slice=None*, *verbose=None*)
Mask to indicate whether a shell is zero on grid

Args: mol : an instance of `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 `Logger`] No effects.

Returns: 2D mask array of shape (N,nbas), where N is the number of grids, nbas is the number of shells.

pyscf.dft.gen_grid.**gen_atomic_grids**(*mol*, *atom_grid={}*, *radi_method=<function gauss_chebyshev>*, *level=3*, *prune=<function nwchem_prune>*)

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.

pyscf.dft.gen_grid.**gen_partition**(*mol*, *atom_grids_tab*, *radii_adjust=None*, *atomic_radii=array([0., 0.66140414, 2.64561657, 2.74010288, 1.98421243, 1.60626721, 1.32280829, 1.22832198, 1.13383567, 0.94486306, 2.83458919, 3.40150702, 2.83458919, 2.36215766, 2.07869874, 1.88972612, 1.88972612, 1.88972612, 3.40150702, 4.15739747, 3.40150702, 3.0235618, 2.64561657, 2.55113027, 2.64561657, 2.64561657, 2.64561657, 2.55113027, 2.55113027, 2.55113027, 2.55113027, 2.45664396, 2.36215766, 2.17318504, 2.17318504, 2.17318504, 3.59047964, 4.44085639, 3.77945225, 3.40150702, 2.92907549, 2.74010288, 2.74010288, 2.55113027, 2.45664396, 2.55113027, 2.64561657, 3.0235618, 2.92907549, 2.92907549, 2.74010288, 2.74010288, 2.64561657, 2.64561657, 3.96842486, 4.91328792, 4.06291117, 3.68496594, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.40150702, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 2.92907549, 2.74010288, 2.55113027, 2.55113027, 2.45664396, 2.55113027, 2.55113027, 2.55113027, 2.83458919, 3.59047964, 3.40150702, 3.0235618, 3.59047964, 2.74010288, 3.96842486, 3.40150702, 4.06291117, 3.68496594, 3.40150702, 3.40150702, 3.30702072])*, *becke_scheme=<function original_becke>*)

Generate the mesh grid coordinates and weights for DFT numerical integration. We can change radii_adjust, becke_scheme functions to generate different meshgrid.

Returns: grid_coord and grid_weight arrays. grid_coord array has shape (N,3); weight 1D array has N elements.

pyscf.dft.gen_grid.**make_mask**(*mol*, *coords*, *relativity=0*, *shls_slice=None*, *verbose=None*)

Mask to indicate whether a shell is zero on grid

Args: mol : an instance of `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 `Logger`] No effects.

Returns: 2D mask array of shape (N,nbas), where N is the number of grids, nbas is the number of shells.

`pyscf.dft.gen_grid.`**`nwchem_prune`**(*nuc*, *rads*, *n_ang*, *radii=array([0., 0.66140414, 2.64561657, 2.74010288, 1.98421243, 1.60626721, 1.32280829, 1.22832198, 1.13383567, 0.94486306, 2.83458919, 3.40150702, 2.83458919, 2.36215766, 2.07869874, 1.88972612, 1.88972612, 1.88972612, 3.40150702, 4.15739747, 3.40150702, 3.0235618, 2.64561657, 2.55113027, 2.64561657, 2.64561657, 2.64561657, 2.55113027, 2.55113027, 2.55113027, 2.55113027, 2.45664396, 2.36215766, 2.17318504, 2.17318504, 2.17318504, 3.59047964, 4.44085639, 3.77945225, 3.40150702, 2.92907549, 2.74010288, 2.74010288, 2.55113027, 2.45664396, 2.55113027, 2.64561657, 3.0235618, 2.92907549, 2.92907549, 2.74010288, 2.74010288, 2.64561657, 2.64561657, 3.96842486, 4.91328792, 4.06291117, 3.68496594, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.49599333, 3.40150702, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 3.30702072, 2.92907549, 2.74010288, 2.55113027, 2.55113027, 2.45664396, 2.55113027, 2.55113027, 2.55113027, 2.83458919, 3.59047964, 3.40150702, 3.0235618, 3.59047964, 2.74010288, 3.96842486, 3.40150702, 4.06291117, 3.68496594, 3.40150702, 3.40150702, 3.30702072])*)

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.

pyscf.dft.gen_grid.**original_becke**(*g*)
Becke, JCP, 88, 2547 (1988)

pyscf.dft.gen_grid.**sg1_prune**(*nuc*, *rads*, *n_ang*, *radii=array([0., 1., 0.5882, 3.0769, 2.0513, 1.5385, 1.2308, 1.0256, 0.8791, 0.7692, 0.6838, 4.0909, 3.1579, 2.5714, 2.1687, 1.875, 1.6514, 1.4754, 1.3333])*)
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.

pyscf.dft.gen_grid.**stratmann**(*g*)
Stratmann, Scuseria, Frisch. CPL, 257, 213 (1996)

pyscf.dft.gen_grid.**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.

pyscf.dft.numint.**cache_xc_kernel**(*ni*, *mol*, *grids*, *xc_code*, *mo_coeff*, *mo_occ*, *spin=0*, *max_memory=2000*)
Compute the 0th order density, Vxc and fxc. They can be used in TDDFT, DFT hessian module etc.

pyscf.dft.numint.**eval_ao**(*mol*, *coords*, *deriv=0*, *shls_slice=None*, *non0tab=None*, *out=None*, *verbose=None*)
Evaluate AO function value on the given grids.

Args: mol : an instance of `Mole`

coords [2D array, shape (N,3)] The coordinates of the grids.

Kwargs:

deriv [int] AO derivative order. It affects the shape of the return array. If deriv=0, the returned AO values are stored in a (N,nao) array. Otherwise the AO values are stored in an array of shape (M,N,nao). Here N is the number of grids, nao is the number of AO functions, M is the size associated to the derivative deriv.

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.

non0tab [2D bool array] mask array to indicate whether the AO values are zero. The mask array can be obtained by calling `make_mask()`

out [ndarray] If provided, results are written into this array.

verbose [int or object of `Logger`] No effects.

Returns: 2D array of shape (N,nao) for AO values if deriv = 0. Or 3D array of shape (:,N,nao) for AO values and AO derivatives if deriv > 0. In the 3D array, the first (N,nao) elements are the AO values, followed by (3,N,nao) for x,y,z compoents; Then 2nd derivatives (6,N,nao) for xx, xy, xz, yy, yz, zz; Then 3rd derivatives (10,N,nao) for xxx, xxy, xxz, xyy, xyz, xzz, yyy, yyz, yzz, zzz; ...

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
>>> coords = numpy.random.random((100,3))  # 100 random points
>>> ao_value = eval_ao(mol, coords)
>>> print(ao_value.shape)
(100, 24)
>>> ao_value = eval_ao(mol, coords, deriv=1, shls_slice=(1,4))
>>> print(ao_value.shape)
(4, 100, 7)
>>> ao_value = eval_ao(mol, coords, deriv=2, shls_slice=(1,4))
>>> print(ao_value.shape)
(10, 100, 7)
```

pyscf.dft.numint.**eval_mat**(*mol*, *ao*, *weight*, *rho*, *vxc*, *non0tab=None*, *xctype='LDA'*, *spin=0*, *verbose=None*)

Calculate XC potential matrix.

Args: mol : an instance of `Mole`

ao [([4/10,] ngrids, nao) ndarray] 2D array of shape (N,nao) for LDA, 3D array of shape (4,N,nao) for GGA or (10,N,nao) for meta-GGA. N is the number of grids, nao is the number of AO functions. If xctype is GGA, ao[0] is AO value and ao[1:3] are the real space gradients. If xctype is meta-GGA, ao[4:10] are second derivatives of ao values.

weight [1D array] Integral weights on grids.

rho [([4/6,] ngrids) ndarray] Shape of *((,N)) for electron density (and derivatives) if spin = 0; Shape of ((*,N),(,*N)) for alpha/beta electron density (and derivatives) if spin > 0; where N is number of grids. rho (*,N) are ordered as (den,grad_x,grad_y,grad_z,laplacian,tau) where grad_x = d/dx den, laplacian = nabla^2 den, tau = 1/2(nabla f)^2 In spin unrestricted case, rho is ((den_u,grad_xu,grad_yu,grad_zu,laplacian_u,tau_u)

(den_d,grad_xd,grad_yd,grad_zd,laplacian_d,tau_d))

vxc [([4,] ngrids) ndarray] XC potential value on each grid = (vrho, vsigma, vlapl, vtau) vsigma is GGA potential value on each grid. If the kwarg spin is not 0, a list [vsigma_uu,vsigma_ud] is required.

Kwargs:

xctype [str] LDA/GGA/mGGA. It affects the shape of *ao* and *rho*

non0tab [2D bool array] mask array to indicate whether the AO values are zero. The mask array can be obtained by calling `make_mask()`

spin [int] If not 0, the matrix is contracted with the spin non-degenerated UKS formula

Returns: XC potential matrix in 2D array of shape (nao,nao) where nao is the number of AO functions.

pyscf.dft.numint.**eval_rho**(*mol*, *ao*, *dm*, *non0tab=None*, *xctype='LDA'*, *hermi=0*, *verbose=None*)

Calculate the electron density for LDA functional, and the density derivatives for GGA functional.

Args: mol : an instance of `Mole`

ao [2D array of shape (N,nao) for LDA, 3D array of shape (4,N,nao) for GGA] or (5,N,nao) for meta-GGA. N is the number of grids, nao is the number of AO functions. If xctype is GGA, ao[0] is AO value and ao[1:3] are the AO gradients. If xctype is meta-GGA, ao[4:10] are second derivatives of ao values.

dm [2D array] Density matrix

Kwargs:

non0tab [2D bool array] mask array to indicate whether the AO values are zero. The mask array can be obtained by calling `make_mask()`

xctype [str] LDA/GGA/mGGA. It affects the shape of the return density.

hermi [bool] dm is hermitian or not

verbose [int or object of `Logger`] No effects.

Returns: 1D array of size N to store electron density if xctype = LDA; 2D array of (4,N) to store density and "density derivatives" for x,y,z components if xctype = GGA; (6,N) array for meta-GGA, where last two rows are nabla^2 rho and tau = 1/2(nabla f)^2

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
>>> coords = numpy.random.random((100,3))  # 100 random points
>>> ao_value = eval_ao(mol, coords, deriv=0)
>>> dm = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> dm = dm + dm.T
>>> rho, dx_rho, dy_rho, dz_rho = eval_rho(mol, ao, dm, xctype='LDA')
```

pyscf.dft.numint.**eval_rho2**(*mol*, *ao*, *mo_coeff*, *mo_occ*, *non0tab=None*, *xctype='LDA'*, *verbose=None*)

Calculate the electron density for LDA functional, and the density derivatives for GGA functional. This function has the same functionality as *eval_rho()* except that the density are evaluated based on orbital coefficients and orbital occupancy. It is more efficient than *eval_rho()* in most scenario.

Args: mol : an instance of `Mole`

ao [2D array of shape (N,nao) for LDA, 3D array of shape (4,N,nao) for GGA] or (5,N,nao) for meta-GGA. N is the number of grids, nao is the number of AO functions. If xctype is GGA, ao[0] is AO value and ao[1:3] are the AO gradients. If xctype is meta-GGA, ao[4:10] are second derivatives of ao values.

dm [2D array] Density matrix

Kwargs:

non0tab [2D bool array] mask array to indicate whether the AO values are zero. The mask array can be obtained by calling `make_mask()`

xctype [str] LDA/GGA/mGGA. It affects the shape of the return density.

verbose [int or object of `Logger`] No effects.

Returns: 1D array of size N to store electron density if xctype = LDA; 2D array of (4,N) to store density and "density derivatives" for x,y,z components if xctype = GGA; (6,N) array for meta-GGA, where last two rows are nabla^2 rho and tau = 1/2(nabla f)^2

pyscf.dft.numint.**large_rho_indices**(*ni*, *mol*, *dm*, *grids*, *cutoff=1e-10*, *max_memory=2000*)

Indices of density which are larger than given cutoff

pyscf.dft.numint.**nr_fxc**(*mol*, *grids*, *xc_code*, *dm0*, *dms*, *spin=0*, *relativity=0*, *hermi=0*, *rho0=None*, *vxc=None*, *fxc=None*, *max_memory=2000*, *verbose=None*)

Contract XC kernel matrix with given density matrices

... math:

```
a_{pq} = f_{pq,rs} * x_{rs}
```

pyscf.dft.numint.**nr_rks**(*ni*, *mol*, *grids*, *xc_code*, *dms*, *relativity=0*, *hermi=0*, *max_memory=2000*, *verbose=None*)

Calculate RKS XC functional and potential matrix on given meshgrids for a set of density matrices

Args: ni : an instance of _NumInt

mol : an instance of Mole

grids [an instance of Grids] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.

xc_code [str] XC functional description. See parse_xc() of pyscf/dft/libxc.py for more details.

dms [2D array a list of 2D arrays] Density matrix or multiple density matrices

Kwargs:

hermi [int] Input density matrices symmetric or not

max_memory [int or float] The maximum size of cache to use (in MB).

Returns: nelec, excsum, vmat. nelec is the number of electrons generated by numerical integration. excsum is the XC functional value. vmat is the XC potential matrix in 2D array of shape (nao,nao) where nao is the number of AO functions.

Examples:

```
>>> from pyscf import gto, dft
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> grids = dft.gen_grid.Grids(mol)
>>> grids.coords = numpy.random.random((100,3))  # 100 random points
>>> grids.weights = numpy.random.random(100)
>>> dm = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> nelec, exc, vxc = dft.numint.nr_vxc(mol, grids, 'lda,vwn', dm)
```

pyscf.dft.numint.**nr_rks_fxc**(*ni*, *mol*, *grids*, *xc_code*, *dm0*, *dms*, *relativity=0*, *hermi=0*, *rho0=None*, *vxc=None*, *fxc=None*, *max_memory=2000*, *verbose=None*)

Contract RKS XC (singlet hessian) kernel matrix with given density matrices

Args: ni : an instance of _NumInt

mol : an instance of Mole

grids [an instance of Grids] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.

xc_code [str] XC functional description. See parse_xc() of pyscf/dft/libxc.py for more details.

dms [2D array a list of 2D arrays] Density matrix or multiple density matrices

Kwargs:

hermi [int] Input density matrices symmetric or not

max_memory [int or float] The maximum size of cache to use (in MB).

rho0 [float array] Zero-order density (and density derivative for GGA). Giving kwargs rho0, vxc and fxc to improve better performance.

vxc [float array] First order XC derivatives

fxc [float array] Second order XC derivatives

Returns: nelec, excsum, vmat. nelec is the number of electrons generated by numerical integration. excsum is the XC functional value. vmat is the XC potential matrix in 2D array of shape (nao,nao) where nao is the number of AO functions.

Examples:

pyscf.dft.numint.**nr_rks_fxc_st**(*ni*, *mol*, *grids*, *xc_code*, *dm0*, *dms_alpha*, *relativity=0*, *singlet=True*, *rho0=None*, *vxc=None*, *fxc=None*, *max_memory=2000*, *verbose=None*)

Associated to singlet or triplet Hessian Note the difference to nr_rks_fxc, dms_alpha is the response density matrices of alpha spin, alpha+/-beta DM is applied due to singlet/triplet coupling

Ref. CPL, 256, 454

pyscf.dft.numint.**nr_rks_vxc**(*ni*, *mol*, *grids*, *xc_code*, *dms*, *relativity=0*, *hermi=0*, *max_memory=2000*, *verbose=None*)

Calculate RKS XC functional and potential matrix on given meshgrids for a set of density matrices

Args: ni : an instance of _NumInt

mol : an instance of Mole

grids [an instance of Grids] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.

xc_code [str] XC functional description. See parse_xc() of pyscf/dft/libxc.py for more details.

dms [2D array a list of 2D arrays] Density matrix or multiple density matrices

Kwargs:

hermi [int] Input density matrices symmetric or not

max_memory [int or float] The maximum size of cache to use (in MB).

Returns: nelec, excsum, vmat. nelec is the number of electrons generated by numerical integration. excsum is the XC functional value. vmat is the XC potential matrix in 2D array of shape (nao,nao) where nao is the number of AO functions.

Examples:

```
>>> from pyscf import gto, dft
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1.1')
>>> grids = dft.gen_grid.Grids(mol)
>>> grids.coords = numpy.random.random((100,3))  # 100 random points
>>> grids.weights = numpy.random.random(100)
>>> dm = numpy.random.random((mol.nao_nr(),mol.nao_nr()))
>>> nelec, exc, vxc = dft.numint.nr_vxc(mol, grids, 'lda,vwn', dm)
```

pyscf.dft.numint.**nr_uks**(*ni*, *mol*, *grids*, *xc_code*, *dms*, *relativity=0*, *hermi=0*, *max_memory=2000*, *verbose=None*)

Calculate UKS XC functional and potential matrix on given meshgrids for a set of density matrices

Args: mol : an instance of Mole

grids [an instance of Grids] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.

xc_code [str] XC functional description. See `parse_xc()` of pyscf/dft/libxc.py for more details.

dms [a list of 2D arrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwargs:

hermi [int] Input density matrices symmetric or not

max_memory [int or float] The maximum size of cache to use (in MB).

Returns: nelec, excsum, vmat. nelec is the number of (alpha,beta) electrons generated by numerical integration. excsum is the XC functional value. vmat is the XC potential matrix for (alpha,beta) spin.

`pyscf.dft.numint.`**`nr_uks_fxc`**(*ni*, *mol*, *grids*, *xc_code*, *dm0*, *dms*, *relativity=0*, *hermi=0*, *rho0=None*, *vxc=None*, *fxc=None*, *max_memory=2000*, *verbose=None*)

Contract UKS XC kernel matrix with given density matrices

Args: ni : an instance of `_NumInt`

mol : an instance of `Mole`

grids [an instance of `Grids`] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.

xc_code [str] XC functional description. See `parse_xc()` of pyscf/dft/libxc.py for more details.

dms [2D array a list of 2D arrays] Density matrix or multiple density matrices

Kwargs:

hermi [int] Input density matrices symmetric or not

max_memory [int or float] The maximum size of cache to use (in MB).

rho0 [float array] Zero-order density (and density derivative for GGA). Giving kwargs rho0, vxc and fxc to improve better performance.

vxc [float array] First order XC derivatives

fxc [float array] Second order XC derivatives

Returns: nelec, excsum, vmat. nelec is the number of electrons generated by numerical integration. excsum is the XC functional value. vmat is the XC potential matrix in 2D array of shape (nao,nao) where nao is the number of AO functions.

Examples:

`pyscf.dft.numint.`**`nr_uks_vxc`**(*ni*, *mol*, *grids*, *xc_code*, *dms*, *relativity=0*, *hermi=0*, *max_memory=2000*, *verbose=None*)

Calculate UKS XC functional and potential matrix on given meshgrids for a set of density matrices

Args: mol : an instance of `Mole`

grids [an instance of `Grids`] grids.coords and grids.weights are needed for coordinates and weights of meshgrids.

xc_code [str] XC functional description. See `parse_xc()` of pyscf/dft/libxc.py for more details.

dms [a list of 2D arrays] A list of density matrices, stored as (alpha,alpha,...,beta,beta,...)

Kwargs:

hermi [int] Input density matrices symmetric or not

max_memory [int or float] The maximum size of cache to use (in MB).

Returns: nelec, excsum, vmat. nelec is the number of (alpha,beta) electrons generated by numerical integration. excsum is the XC functional value. vmat is the XC potential matrix for (alpha,beta) spin.

XC functional, the interface to libxc (http://www.tddft.org/programs/octopus/wiki/index.php/Libxc)

pyscf.dft.libxc.**define_xc**(*ni*, *description*)

Define XC functional. See also *eval_xc()* for the rules of input description.

Args: ni : an instance of _NumInt

description [str] A string to describe the linear combination of different XC functionals. The X and C functional are separated by comma like '.8*LDA+.2*B86,VWN'. If "HF" was appeared in the string, it stands for the exact exchange.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
>>> mf = dft.RKS(mol)
>>> define_xc_(mf._numint, '.2*HF + .08*LDA + .72*B88, .81*LYP + .19*VWN')
>>> mf.kernel()
-76.3783361189611
>>> define_xc_(mf._numint, 'LDA*.08 + .72*B88 + .2*HF, .81*LYP + .19*VWN')
>>> mf.kernel()
-76.3783361189611
>>> def eval_xc(xc_code, rho, *args, **kwargs):
...     exc = 0.01 * rho**2
...     vrho = 0.01 * 2 * rho
...     vxc = (vrho, None, None, None)
...     fxc = None  # 2nd order functional derivative
...     kxc = None  # 3rd order functional derivative
...     return exc, vxc, fxc, kxc
>>> define_xc_(mf._numint, eval_xc, xctype='LDA')
>>> mf.kernel()
48.8525211046668
```

pyscf.dft.libxc.**define_xc_**(*ni*, *description*, *xctype='LDA'*, *hyb=0*)

Define XC functional. See also *eval_xc()* for the rules of input description.

Args: ni : an instance of _NumInt

description [str] A string to describe the linear combination of different XC functionals. The X and C functional are separated by comma like '.8*LDA+.2*B86,VWN'. If "HF" was appeared in the string, it stands for the exact exchange.

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz')
>>> mf = dft.RKS(mol)
>>> define_xc_(mf._numint, '.2*HF + .08*LDA + .72*B88, .81*LYP + .19*VWN')
>>> mf.kernel()
-76.3783361189611
>>> define_xc_(mf._numint, 'LDA*.08 + .72*B88 + .2*HF, .81*LYP + .19*VWN')
>>> mf.kernel()
-76.3783361189611
>>> def eval_xc(xc_code, rho, *args, **kwargs):
...     exc = 0.01 * rho**2
...     vrho = 0.01 * 2 * rho
...     vxc = (vrho, None, None, None)
...     fxc = None  # 2nd order functional derivative
...     kxc = None  # 3rd order functional derivative
...     return exc, vxc, fxc, kxc
```

```
>>> define_xc_(mf._numint, eval_xc, xctype='LDA')
>>> mf.kernel()
48.8525211046668
```

pyscf.dft.libxc.**eval_xc**(*xc_code*, *rho*, *spin=0*, *relativity=0*, *deriv=1*, *verbose=None*)

Interface to call libxc library to evaluate XC functional, potential and functional derivatives.

- The given functional xc_code must be a one-line string.
- The functional xc_code is case-insensitive.
- The functional xc_code string has two parts, separated by ",". The first part describes the exchange functional, the second is the correlation functional.
  - If "," not appeared in string, the entire string is considered as X functional.
  - To neglect X functional (just apply C functional), leave blank in the first part, eg description=',vwn' for pure VWN functional
- The functional name can be placed in arbitrary order. Two name needs to be separated by operators "+" or "-". Blank spaces are ignored. NOTE the parser only reads operators "+" "-" "*". / is not in support.
- A functional name is associated with one factor. If the factor is not given, it is assumed equaling 1.
- String "HF" stands for exact exchange (HF K matrix). It is allowed to put in C functional part.
- Be careful with the libxc convention on GGA functional, in which the LDA contribution is included.

Args:

xc_code [str] A string to describe the linear combination of different XC functionals. The X and C functional are separated by comma like '.8*LDA+.2*B86,VWN'. If "HF" was appeared in the string, it stands for the exact exchange.

rho [ndarray] Shape of *((,N)) for electron density (and derivatives) if spin = 0; Shape of ((*,N),(*,N)) for alpha/beta electron density (and derivatives) if spin > 0; where N is number of grids. rho (*,N) are ordered as (den,grad_x,grad_y,grad_z,laplacian,tau) where grad_x = d/dx den, laplacian = nabla^2 den, tau = 1/2(nabla f)^2 In spin unrestricted case, rho is ((den_u,grad_xu,grad_yu,grad_zu,laplacian_u,tau_u)

(den_d,grad_xd,grad_yd,grad_zd,laplacian_d,tau_d))

Kwargs:

spin [int] spin polarized if spin > 0

relativity [int] No effects.

verbose [int or object of `Logger`] No effects.

Returns: ex, vxc, fxc, kxc

where

- vxc = (vrho, vsigma, vlapl, vtau) for restricted case
- vxc for unrestricted case | vrho[:,2] = (u, d) | vsigma[:,3] = (uu, ud, dd) | vlapl[:,2] = (u, d) | vtau[:,2] = (u, d)
- fxc for restricted case: (v2rho2, v2rhosigma, v2sigma2, v2lapl2, vtau2, v2rholapl, v2rhotau, v2lapltau, v2sigmalapl, v2sigmatau)
- fxc for unrestricted case: | v2rho2[:,3] = (u_u, u_d, d_d) | v2rhosigma[:,6] = (u_uu, u_ud, u_dd, d_uu, d_ud, d_dd) | v2sigma2[:,6] = (uu_uu, uu_ud, uu_dd, ud_ud, ud_dd, dd_dd) | v2lapl2[:,3] | vtau2[:,3] | v2rholapl[:,4] | v2rhotau[:,4] | v2lapltau[:,4] | v2sigmalapl[:,6] | v2sigmatau[:,6]

- kxc for restricted case: (v3rho3, v3rho2sigma, v3rhosigma2, v3sigma3)
- kxc for unrestricted case: | v3rho3[:,4] = (u_u_u, u_u_d, u_d_d, d_d_d) | v3rho2sigma[:,9] = (u_u_uu, u_u_ud, u_u_dd, u_d_uu, u_d_ud, u_d_dd, d_d_uu, d_d_ud, d_d_dd) | v3rhosigma2[:,12] = (u_uu_uu, u_uu_ud, u_uu_dd, u_ud_ud, u_ud_dd, u_dd_dd, d_uu_uu, d_uu_ud, d_uu_dd, d_ud_ud, d_ud_dd, d_dd_dd) | v3sigma3[:,10] = (uu_uu_uu, uu_uu_ud, uu_uu_dd, uu_ud_ud, uu_ud_dd, uu_dd_dd, ud_ud_ud, ud_ud_dd, ud_dd_dd, dd_dd_dd)

see also libxc_itrf.c

pyscf.dft.libxc.**hybrid_coeff**(*xc_code*, *spin=0*)
Support recursively defining hybrid functional

pyscf.dft.libxc.**parse_xc**(*description*)
Rules to input functional description:

•The given functional description must be a one-line string.

•The functional description is case-insensitive.

•The functional description string has two parts, separated by ",". The first part describes the exchange functional, the second is the correlation functional.

–If "," was not appeared in string, the entire string is considered as X functional.

–To neglect X functional (just apply C functional), leave blank in the first part, eg description=',vwn' for pure VWN functional

•The functional name can be placed in arbitrary order. Two names need to be separated by operators "+" or "-". Blank spaces are ignored. NOTE the parser only reads operators "+" "-" "*". "/" is not supported.

•A functional name is associated with one factor. If the factor is not given, it is assumed to equal 1.

•String "HF" stands for exact exchange (HF K matrix). It is allowed to in the C functional part.

•Be careful with the libxc convention on GGA functional, in which the LDA contribution is included.

pyscf.dft.libxc.**parse_xc_name**(*xc_name='LDA*, *VWN'*)
Convert the XC functional name to libxc library internal ID.

1.13 tddft — Time dependent density functional theory

1.13.1 TDHF

pyscf.tddft.rhf.**gen_tda_hop**(*mf*, *fock_ao=None*, *singlet=True*, *wfnsym=None*, *max_memory=2000*)
(A+B)x

Kwargs:

wfnsym [int] Point group symmetry for excited CIS wavefunction.

1.13.2 TDDFT

class pyscf.tddft.rks.**TDDFTNoHybrid**(*mf*)
Solve (A-B)(A+B)(X+Y) = (X+Y)w^2

kernel(*x0=None*)
TDDFT diagonalization solver

1.14 cc — Coupled cluster

The *cc* module implements the coupled cluster (CC) model to compute energies, analytical nuclear gradients, density matrices, excited states, and relevant properties.

To compute the CC energy, one first needs to perform a mean-field calculation using the mean-field module *scf*. The mean-field object defines the Hamiltonian and the problem size, which are used to initialize the CC object:

```
from pyscf import gto, scf, cc
mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz')
mf = scf.RHF(mol).run()
mycc = cc.CCSD(mf)
mycc.kernel()
```

Unrelaxed density matrices are evaluated in the MO basis:

```
dm1 = mycc.make_rdm1()
dm2 = mycc.make_rdm2()
```

The CCSD(T) energy can be obtained by:

```
from pyscf.cc import ccsd_t
print(ccsd_t.kernel(mycc, mycc.ao2mo())[0])
```

Gradients are available:

```
from pyscf.cc import ccsd_grad
from pyscf import grad
grad_e = ccsd_grad.kernel(mycc)
grad_n = grad.grad_nuc(mol)
grad = grad_e + grad_nuc
```

Excited states can be calculated with ionization potential (IP), electron affinity (EA), and electronic excitation (EE) equation-of-motion (EOM) CCSD:

```
mycc = cc.RCCSD(mf)
mycc.kernel()
e_ip, c_ip = mycc.ipccsd(nroots=1)
e_ea, c_ea = mycc.eaccsd(nroots=1)
e_ee, c_ee = mycc.eeccsd(nroots=1)

mycc = cc.UCCSD(mf)
mycc.kernel()
e_ip, c_ip = mycc.ipccsd(nroots=1)
e_ea, c_ea = mycc.eaccsd(nroots=1)
e_ee, c_ee = mycc.eeccsd(nroots=1)
```

All CC methods have two implementations. One is simple and highly readable (suffixed by `_slow` in the filename) and the other is extensively optimized for computational efficiency. All code in the `_slow` versions is structured as close as possible to the formulas documented in the literature. Pure Python/numpy data structures and functions are used so that explicit memory management is avoided. It is easy to make modifications or develop new methods based on the slow implementations.

The computationally efficient (outcore) version is the default implementation for the CC module. In this implementation, the CPU usage, memory footprint, memory efficiency, and IO overhead are carefully considered. To keep a small memory footprint, most integral tensors are stored on disk. IO is one of the main bottlenecks in this implementation. Two techniques are used to reduce the IO overhead. One is the asynchronized IO to overlap the computation

and reading/writing of the 4-index tensors. The other is AO-driven for the contraction of T2 and `(vv|vv)` integrals in CCSD and CCSD-lambda functions. These techniques allow the CC module to efficiently handle medium-sized systems. In a test system with 25 occupied orbitals and 1500 virtual orbitals, each CCSD iteration takes about 2.5 hours. The program does not automatically switch to AO-driven CCSD for large systems. The user must manually set the `direct` attribute to enable an AO-driven CCSD calculation:

```
mycc = cc.CCSD(mf)
mycc.direct = True
mycc.kernel()
```

Some of the CC methods have an efficient incore implementation, where all tensors are held in memory. The incore implementation reduces the IO overhead and optimizes certain formulas to gain the best FLOPS. It is about 30% faster than the outcore implementation. Depending on the available memory, the incore code can be used for systems with up to approximately 250 orbitals.

Point group symmetry is not considered in the CCSD programs, but it is used in the CCSD(T) code to gain the best performance.

Arbitrary frozen orbitals (not limited to frozen core) are supported by the CCSD, CCSD(T), density matrices, and EOM-CCSD modules, but not in the analytical CCSD gradient module.

1.14.1 Examples

This section documents some examples about how to effectively use the CCSD module, and how to incorporate the CCSD solver with other PySCF functions to perform advanced simulations. For a complete list of CC examples, see `pyscf/examples/cc`.

A general solver for customized Hamiltonian

The CC module is not limited to molecular systems. The program is implemented as a general solver for arbitrary Hamiltonians. It allows users to overwrite the default molecular Hamiltonian with their own effective Hamiltonians. In this example, we create a Hubbard model and feed its Hamiltonian to the CCSD module.

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

'''
Six-site 1D U/t=2 Hubbard-like model system with PBC at half filling.
The model is gapped at the mean-field level
'''

import numpy
from pyscf import gto, scf, ao2mo, cc

mol = gto.M(verbose=4)
n = 6
mol.nelectron = n
# Setting incore_anyway=True to ensure the customized Hamiltonian (the _eri
# attribute) being used in post-HF calculations.  Without this parameter, some
# post-HF method may ignore the customized Hamiltonian if memory is not
# enough.
mol.incore_anyway = True

h1 = numpy.zeros((n,n))
for i in range(n-1):
    h1[i,i+1] = h1[i+1,i] = -1.0
h1[n-1,0] = h1[0,n-1] = -1.0
```

```
eri = numpy.zeros((n,n,n,n))
for i in range(n):
    eri[i,i,i,i] = 2.0

mf = scf.RHF(mol)
mf.get_hcore = lambda *args: h1
mf.get_ovlp = lambda *args: numpy.eye(n)
mf._eri = ao2mo.restore(8, eri, n)
mf.kernel()


# In PySCF, the customized Hamiltonian needs to be created once in mf object.
# The Hamiltonian will be used everywhere whenever possible.  Here, the model
# Hamiltonian is passed to CCSD object via the mf object.

mycc = cc.RCCSD(mf)
mycc.kernel()
e,v = mycc.ipccsd(nroots=3)
print(e)
```

Using CCSD as CASCI active space solver

CCSD program can be wrapped as a Full CI solver, which can be combined with the CASCI solver to approximate the multi-configuration calculation.

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

'''
Using the CCSD method as the active space solver to compute an approximate
CASCI energy.

A wrapper is required to adapt the CCSD solver to CASCI fcisolver interface.
Inside the wrapper function, the CCSD code is the same as the example
40-ccsd_with_given_hamiltonian.py
'''

import numpy
from pyscf import gto, scf, cc, ao2mo, mcscf

class AsFCISolver(object):
    def __init__(self):
        self.mycc = None

    def kernel(self, h1, h2, norb, nelec, ci0=None, ecore=0, **kwargs):
        fakemol = gto.M(verbose=0)
        nelec = numpy.sum(nelec)
        fakemol.nelectron = nelec
        fake_hf = scf.RHF(fakemol)
        fake_hf._eri = ao2mo.restore(8, h2, norb)
        fake_hf.get_hcore = lambda *args: h1
        fake_hf.get_ovlp = lambda *args: numpy.eye(norb)
        fake_hf.kernel()
        self.mycc = cc.CCSD(fake_hf)
        self.eris = self.mycc.ao2mo()
        e_corr, t1, t2 = self.mycc.kernel(eris=self.eris)
        l1, l2 = self.mycc.solve_lambda(t1, t2, eris=self.eris)
```

```
        e = fake_hf.e_tot + e_corr
        return e+ecore, [t1,t2,l1,l2]

    def make_rdm1(self, fake_ci, norb, nelec):
        mo = self.mycc.mo_coeff
        t1, t2, l1, l2 = fake_ci
        dm1 = reduce(numpy.dot, (mo, self.mycc.make_rdm1(t1, t2, l1, l2), mo.T))
        return dm1

    def make_rdm12(self, fake_ci, norb, nelec):
        mo = self.mycc.mo_coeff
        nmo = mo.shape[1]
        t1, t2, l1, l2 = fake_ci
        dm2 = self.mycc.make_rdm2(t1, t2, l1, l2)
        dm2 = numpy.dot(mo, dm2.reshape(nmo,-1))
        dm2 = numpy.dot(dm2.reshape(-1,nmo), mo.T)
        dm2 = dm2.reshape([nmo]*4).transpose(2,3,0,1)
        dm2 = numpy.dot(mo, dm2.reshape(nmo,-1))
        dm2 = numpy.dot(dm2.reshape(-1,nmo), mo.T)
        dm2 = dm2.reshape([nmo]*4)
        return self.make_rdm1(fake_ci, norb, nelec), dm2

    def spin_square(self, fake_ci, norb, nelec):
        return 0, 1

mol = gto.M(atom = 'H 0 0 0; F 0 0 1.2',
            basis = 'ccpvdz',
            verbose = 4)
mf = scf.RHF(mol).run()
norb = mf.mo_coeff.shape[1]
nelec = mol.nelectron
mc = mcscf.CASCI(mf, norb, nelec)
mc.fcisolver = AsFCISolver()
mc.kernel()
```

Gamma point CCSD with Periodic boundary condition

Integrals in Gamma point of periodic Hartree-Fock calculation are all real. You can feed the integrals into any pyscf molecular module using the same operations as the above example. However, the interface between PBC code and molecular code are more compatible. You can treat the crystal object and the molecule object in the same manner. In this example, you can pass the PBC mean field method to CC module to have the gamma point CCSD correlation.

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

'''
Gamma point post-HF calculation needs only real integrals.
Methods implemented in finite-size system can be directly used here without
any modification.
'''

import numpy
from pyscf.pbc import gto, scf

cell = gto.M(
    a = numpy.eye(3)*3.5668,
```

```
    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 = '6-31g',
    verbose = 4,
)

mf = scf.RHF(cell).density_fit()
mf.with_df.gs = [5]*3
mf.kernel()

#
# Import CC, TDDFT moduel from the molecular implementations
#
from pyscf import cc, tddft
mycc = cc.CCSD(mf)
mycc.kernel()

mytd = tddft.TDHF(mf)
mytd.nstates = 5
mytd.kernel()
```

CCSD with truncated MOs to avoid linear dependency

It is common to have linear dependence when one wants to systematically enlarge the AO basis set to approach complete basis set limit. The numerical instability usually has noticeable effects on the CCSD convergence. An effective way to remove this negative effects is to truncate the AO sets and allow the MO orbitals being less than AO functions.

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

'''
:func:`scf.addons.remove_linear_dep_` discards the small eigenvalues of overlap
matrix.  This reduces the number of MOs from 50 to 49.  The problem size of
the following CCSD method is 49.
'''

from pyscf import gto, scf, cc
mol = gto.Mole()
mol.atom = [('H', 0, 0, .5*i) for i in range(20)]
mol.basis = 'ccpvdz'
mol.verbose = 4
mol.build()
mf = scf.RHF(mol).run()
mycc = cc.CCSD(mf).run()

mf = scf.addons.remove_linear_dep_(mf).run()
mycc = cc.CCSD(mf).run()
```

Response and un-relaxed CCSD density matrix

CCSD has two kinds of one-particle density matrices. The (second order) un-relaxed density matrix and the (relaxed) response density matrix. The `CCSD.make_rdm1()` function computes the un-relaxed density matrix which is associated to the regular CCSD energy formula. The response density is mainly used to compute the first order response quantities eg the analytical nuclear gradients. It is not recommended to use the response density matrix for population analysis.

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

'''
CCSD density matrix
'''

from pyscf import gto, scf, cc

mol = gto.M(
    atom = 'H 0 0 0; F 0 0 1.1',
    basis = 'ccpvdz')
mf = scf.RHF(mol).run()
mycc = cc.CCSD(mf).run()

#
# CCSD density matrix in MO basis
#
dm1 = mycc.make_rdm1()
dm2 = mycc.make_rdm2()

#
# Relaxed CCSD density matrix in MO basis
#
from pyscf.cc import ccsd_grad
dm1 += ccsd_grad.response_dm1(mycc, mycc.t1, mycc.t2, mycc.l1, mycc.l2)
```

Reusing integrals in CCSD and relevant calculations

By default the CCSD solver and the relevant CCSD lambda solver, CCSD(T), CCSD gradients program generate MO integrals in their own runtime. But in most scenario, the same MO integrals can be generated once and reused in the four modules. To remove the overhead of recomputing MO integrals, the three module support user to feed MO integrals.

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

'''
To avoid recomputing AO to MO integral transformation, integrals for CCSD,
CCSD(T), CCSD lambda equation etc can be reused.
'''

from pyscf import gto, scf, cc
```

```
mol = gto.M(verbose = 4,
    atom = 'H 0 0 0; F 0 0 1.1',
    basis = 'ccpvdz')

mf = scf.RHF(mol).run()
mycc = cc.CCSD(mf)

#
# CCSD module allows you feed MO integrals
#
eris = mycc.ao2mo()
mycc.kernel(eris=eris)

#
# The same MO integrals can be used in CCSD lambda equation
#
mycc.solve_lambda(eris=eris)

#
# CCSD(T) module requires the same integrals used by CCSD module
#
from pyscf.cc import ccsd_t
ccsd_t.kernel(mycc, eris=eris)

#
# CCSD gradients need regular MO integrals to solve the relaxed 1-particle
# density matrix
#
from pyscf.cc import ccsd_grad
grad_e = ccsd_grad.kernel(mycc, eris=eris)  # The electronic part only
```

Interfering CCSD-DIIS

Restart CCSD

1.14.2 Program reference

cc.ccsd module and CCSD class

The *pyscf.cc.ccsd.CCSD* class is the object to hold the restricted CCSD environment attributes and results. The environment attributes are the parameters to control the runtime behavior of the CCSD module, e.g. the convergence criteria, DIIS parameters, and so on. After the ground state CCSD calculation, correlation energy, `T1` and `T2` amplitudes are stored in the CCSD object. This class supports the calculation of CCSD 1- and 2-particle density matrices.

class `pyscf.cc.ccsd.`**CCSD** (*mf*, *frozen=0*, *mo_coeff=None*, *mo_occ=None*)

restricted CCSD

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`

conv_tol [float] converge threshold. Default is 1e-7.

conv_tol_normt [float] converge threshold for norm(t1,t2). Default is 1e-5.

max_cycle [int] max number of iterations. Default is 50.

diis_space [int] DIIS space size. Default is 6.

diis_start_cycle [int] The step to start DIIS. Default is 0.

direct [bool] AO-direct CCSD. Default is False.

frozen [int or list] If integer is given, the inner-most orbitals are frozen from CC amplitudes. Given the orbital indices (0-based) in a list, both occupied and virtual orbitals can be frozen in CC calculation.

```
>>> mol = gto.M(atom = 'H 0 0 0; F 0 0 1.1', basis = 'ccpvdz')
>>> mf = scf.RHF(mol).run()
>>> # freeze 2 core orbitals
>>> mycc = cc.CCSD(mf).set(frozen = 2).run()
>>> # freeze 2 core orbitals and 3 high lying unoccupied orbitals
>>> mycc.set(frozen = [0,1,16,17,18]).run()
```

Saved results

converged [bool] CCSD converged or not

e_corr [float] CCSD correlation correction

e_tot [float] Total CCSD energy (HF + correlation)

t1, t2 [] T amplitudes t1[i,a], t2[i,j,a,b] (i,j in occ, a,b in virt)

l1, l2 [] Lambda amplitudes l1[i,a], l2[i,j,a,b] (i,j in occ, a,b in virt)

`pyscf.cc.ccsd.`**CC**
alias of *CCSD*

class `pyscf.cc.ccsd.`**CCSD**(*mf*, *frozen=0*, *mo_coeff=None*, *mo_occ=None*)
restricted CCSD

Attributes:

verbose [int] Print level. Default value equals to `Mole.verbose`

max_memory [float or int] Allowed memory in MB. Default value equals to `Mole.max_memory`

conv_tol [float] converge threshold. Default is 1e-7.

conv_tol_normt [float] converge threshold for norm(t1,t2). Default is 1e-5.

max_cycle [int] max number of iterations. Default is 50.

diis_space [int] DIIS space size. Default is 6.

diis_start_cycle [int] The step to start DIIS. Default is 0.

direct [bool] AO-direct CCSD. Default is False.

frozen [int or list] If integer is given, the inner-most orbitals are frozen from CC amplitudes. Given the orbital indices (0-based) in a list, both occupied and virtual orbitals can be frozen in CC calculation.

```
>>> mol = gto.M(atom = 'H 0 0 0; F 0 0 1.1', basis = 'ccpvdz')
>>> mf = scf.RHF(mol).run()
>>> # freeze 2 core orbitals
>>> mycc = cc.CCSD(mf).set(frozen = 2).run()
>>> # freeze 2 core orbitals and 3 high lying unoccupied orbitals
>>> mycc.set(frozen = [0,1,16,17,18]).run()
```

Saved results

converged [bool] CCSD converged or not

e_corr [float] CCSD correlation correction

e_tot [float] Total CCSD energy (HF + correlation)

t1, t2 [] T amplitudes t1[i,a], t2[i,j,a,b] (i,j in occ, a,b in virt)

l1, l2 [] Lambda amplitudes l1[i,a], l2[i,j,a,b] (i,j in occ, a,b in virt)

as_scanner(*cc*)

Generating a scanner/solver for CCSD PES.

The returned solver is a function. This function requires one argument "mol" as input and returns total CCSD energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the CCSD and the underlying SCF objects (conv_tol, max_memory etc) are automatically applied in the solver.

Note scanner has side effects. It may change many underlying objects (_scf, with_df, with_x2c, ...) during calculation.

Examples:

```
>>> from pyscf import gto, scf, cc
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> cc_scanner = cc.CCSD(scf.RHF(mol)).as_scanner()
>>> e_tot, grad = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

energy(*mycc*, *t1*, *t2*, *eris*)

CCSD correlation energy

make_rdm1(*t1=None*, *t2=None*, *l1=None*, *l2=None*)

Un-relaxed 1-particle density matrix in MO space

make_rdm2(*t1=None*, *t2=None*, *l1=None*, *l2=None*)

2-particle density matrix in MO space. The density matrix is stored as

dm2[p,r,q,s] = <p^+ q^+ s r>

pyscf.cc.ccsd.**as_scanner**(*cc*)

Generating a scanner/solver for CCSD PES.

The returned solver is a function. This function requires one argument "mol" as input and returns total CCSD energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the CCSD and the underlying SCF objects (conv_tol, max_memory etc) are automatically applied in the solver.

Note scanner has side effects. It may change many underlying objects (_scf, with_df, with_x2c, ...) during calculation.

Examples:

```
>>> from pyscf import gto, scf, cc
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> cc_scanner = cc.CCSD(scf.RHF(mol)).as_scanner()
>>> e_tot, grad = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

pyscf.cc.ccsd.**energy**(*mycc*, *t1*, *t2*, *eris*)
CCSD correlation energy

cc.rccsd and RCCSD class

pyscf.cc.rccsd.RCCSD is also a class for restricted CCSD calculations, but different to the *pyscf.cc.ccsd.CCSD* class. It uses different formula to compute the ground state CCSD solution. Although slower than the implmentation in the *pyscf.cc.ccsd.CCSD* class, it supports the system with complex integrals. Another difference is that this class supports EOM-CCSD methods, including EOM-IP-CCSD, EOM-EA-CCSD, EOM-EE-CCSD, EOM-SF-CCSD.

class pyscf.cc.rccsd.**RCCSD**(*mf*, *frozen=0*, *mo_coeff=None*, *mo_occ=None*)
restricted CCSD with IP-EOM, EA-EOM, EE-EOM, and SF-EOM capabilities

Ground-state CCSD is performed in optimized ccsd.CCSD and EOM is performed here.

class pyscf.cc.rccsd.**RCCSD**(*mf*, *frozen=0*, *mo_coeff=None*, *mo_occ=None*)
restricted CCSD with IP-EOM, EA-EOM, EE-EOM, and SF-EOM capabilities

Ground-state CCSD is performed in optimized ccsd.CCSD and EOM is performed here.

ccsd(*t1=None*, *t2=None*, *eris=None*, *mbpt2=False*)
Ground-state CCSD.

Kwargs:

mbpt2 [bool] Use one-shot MBPT2 approximation to CCSD.

eaccsd(*nroots=1*, *left=False*, *koopmans=False*, *guess=None*, *partition=None*)
Calculate (N+1)-electron charged excitations via EA-EOM-CCSD.

Kwargs: See ipccd()

eeccsd(*nroots=1*, *koopmans=False*, *guess=None*)
Calculate N-electron neutral excitations via EE-EOM-CCSD.

Kwargs:

nroots [int] Number of roots (eigenvalues) requested

koopmans [bool] Calculate Koopmans'-like (1p1h) excitations only, targeting via overlap.

guess [list of ndarray] List of guess vectors to use for targeting via overlap.

eomsf_ccsd_matvec(*vector*)
Spin flip EOM-CCSD

ipccsd(*nroots=1*, *left=False*, *koopmans=False*, *guess=None*, *partition=None*)
Calculate (N-1)-electron charged excitations via IP-EOM-CCSD.

Kwargs:

nroots [int] Number of roots (eigenvalues) requested

partition [bool or str] Use a matrix-partitioning for the doubles-doubles block. Can be None, 'mp' (Moller-Plesset, i.e. orbital energies on the diagonal), or 'full' (full diagonal elements).

koopmans [bool] Calculate Koopmans'-like (quasiparticle) excitations only, targeting via overlap.

guess [list of ndarray] List of guess vectors to use for targeting via overlap.

pyscf.cc.rccsd.**kernel**(*cc*, *eris*, *t1=None*, *t2=None*, *max_cycle=50*, *tol=1e-08*, *tolnormt=1e-06*, *verbose=4*)
Exactly the same as pyscf.cc.ccsd.kernel, which calls a *local* energy() function.

cc.uccsd and UCCSD class

pyscf.cc.uccsd.UCCSD class supports the CCSD calculation based on UHF wavefunction as well as the ROHF wavefunction. Besides the ground state UCCSD calculation, UCCSD lambda equation, 1-particle and 2-particle density matrices, EOM-IP-CCSD, EOM-EA-CCSD, EOM-EE-CCSD are all available in this class. Note this class does not support complex integrals.

class pyscf.cc.uccsd.**UCCSD**(*mf*, *frozen=0*, *mo_coeff=None*, *mo_occ=None*)

UCCSD with spatial integrals

pyscf.cc.uccsd.**get_umoidx**(*cc*)
Get MO boolean indices for unrestricted reference, accounting for frozen orbs.

pyscf.cc.uccsd.**kernel**(*cc*, *eris*, *t1=None*, *t2=None*, *max_cycle=50*, *tol=1e-08*, *tolnormt=1e-06*, *verbose=4*)
Exactly the same as pyscf.cc.ccsd.kernel, which calls a *local* energy() function.

pyscf.cc.uccsd.**uspatial2spin**(*cc*, *moidx*, *mo_coeff*)
Convert the results of an unrestricted mean-field calculation to spin-orbital form.

Spin-orbital ordering is determined by orbital energy without regard for spin.

Returns:

fock [(nso,nso) ndarray] The Fock matrix in the basis of spin-orbitals

so_coeff [(nao, nso) ndarray] The matrix of spin-orbital coefficients in the AO basis

spin [(nso,) ndarary] The spin (0 or 1) of each spin-orbital

cc.addons

Helper functions for CCSD, RCCSD and UCCSD modules are implemented in cc.addons

pyscf.cc.addons.**spatial2spin**(*tx*, *orbspin=None*)
Convert T1/T2 of spatial orbital representation to T1/T2 of spin-orbital representation

call orbspin_of_sorted_mo_energy to get orbspin

pyscf.cc.addons.**spatial2spinorb**(*tx*, *orbspin=None*)
Convert T1/T2 of spatial orbital representation to T1/T2 of spin-orbital representation

call orbspin_of_sorted_mo_energy to get orbspin

pyscf.cc.addons.**spin2spatial**(*tx*, *orbspin*)
call orbspin_of_sorted_mo_energy to get orbspin

CCSD(T)

CCSD gradients

pyscf.cc.ccsd_grad.**as_scanner**(*cc*)
Generating a scanner/solver for CCSD PES.

The returned solver is a function. This function requires one argument "mol" as input and returns total CCSD energy.

The solver will automatically use the results of last calculation as the initial guess of the new calculation. All parameters assigned in the CCSD and the underlying SCF objects (conv_tol, max_memory etc) are automatically applied in the solver.

Note scanner has side effects. It may change many underlying objects (_scf, with_df, with_x2c, ...) during calculation.

Examples:

```
>>> from pyscf import gto, scf, cc
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1')
>>> cc_scanner = cc.CCSD(scf.RHF(mol)).as_scanner()
>>> e_tot, grad = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.1'))
>>> e_tot, grad = cc_scanner(gto.M(atom='H 0 0 0; F 0 0 1.5'))
```

1.15 ci — Configuration interaction

The *cc* module implements the truncated CI model to compute energy.

1.16 dmrgscf

DMRG program interface.

There are two DMRG program interfaces available:

- Block interface provided the features including the DMRG-CASCI, the 1-step and 2-step DMRG-CASSCF, second order pertubation for dynamic correlation. 1-, 2- and 3-particle density matrices.
- CheMPS2 interface provided the DMRG-CASCI and 2-step DMRG-CASSCF.

Simple usage:

```
>>> from pyscf import gto, scf, mcscf, dmrgscf, mrpt
>>> mol = gto.M(atom='C 0 0 0; C 0 0 1', basis='631g')
>>> mf = scf.RHF(mol).run()
>>> mc = dmrgscf.DMRGSCF(mf, 4, 4)
>>> mc.kernel()
-75.3374492511669
>>> mrpt.NEVPT(mc).compress_approx().kernel()
-0.10474250075684

>>> mc = mcscf.CASSCF(mf, 4, 4)
>>> mc.fcisolver = dmrgscf.CheMPS2(mol)
>>> mc.kernel()
-75.3374492511669
```

Note a few configurations in `/path/to/dmrgscf/settings.py` needs to be made before using the DMRG interface code.

1.16.1 Block

`DMRGCI` is the main object to hold Block input parameters and results. `DMRGSCF()` is a shortcut function quickly setup DMRG-CASSCF calculation. `compress_approx()` initializes the compressed MPS perturber for NEVPT2 calculation.

In DMRGCI object, you can set the following attributes to control Block program:

outputlevel [int] Noise level for Block program output.

maxIter [int] Max DMRG sweeps

approx_maxIter [int] To control the DMRG-CASSCF approximate DMRG solver accuracy.

twodot_to_onedot [int] When to switch from two-dot algroithm to one-dot algroithm.

nroots [int] Number of states in the same irreducible representation to compute.

weights [list of floats] Use this attribute with "nroots" attribute to set state-average calculation.

restart [bool] To control whether to restart a DMRG calculation.

tol [float] DMRG convergence tolerence

maxM [int] Bond dimension

scheduleSweeps, scheduleMaxMs, scheduleTols, scheduleNoises [list] DMRG sweep scheduler. See also Block documentation

wfnsym [str or int] Wave function irrep label or irrep ID

orbsym [list of int] irrep IDs of each orbital

groupname [str] groupname, orbsym together can control whether to employ symmetry in the calculation. "groupname = None and orbsym = []" requires the Block program using C1 symmetry.

1.16.2 CheMPS2

In `CheMPS2`, DMRG calculation can be controlled by:

wfn_irrep
dmrg_states
dmrg_noise
dmrg_e_convergence
dmrg_noise_factor
dmrg_maxiter_noise
dmrg_maxiter_silent

See http://sebwouters.github.io/CheMPS2/index.html for more detail usages of these keywords.

1.17 fciqmcscf

1.18 tools

1.18.1 FCIDUMP

`pyscf.tools.fcidump.`**`from_chkfile`**(*output*, *chkfile*, *tol=1e-15*, *float_format=' %.16g'*)
Read SCF results from PySCF chkfile and transform 1-electron, 2-electron integrals using the SCF orbitals. The transformed integrals is written to FCIDUMP

`pyscf.tools.fcidump.`**`from_integrals`**(*output*, *h1e*, *h2e*, *nmo*, *nelec*, *nuc=0*, *ms=0*, *orbsym=[]*, *tol=1e-15*, *float_format=' %.16g'*)
Convert the given 1-electron and 2-electron integrals to FCIDUMP format

`pyscf.tools.fcidump.`**`read`**(*filename*)
Parse FCIDUMP. Return a dictionary to hold the integrals and parameters with keys: H1, H2, ECORE, NORB, NELEC, MS, ORBSYM, ISYM

1.18.2 Molden

pyscf.tools.molden.**load**(*moldenfile*)
Extract mol and orbitals from molden file

pyscf.tools.molden.**remove_high_l**(*mol*, *mo_coeff=None*)
Remove high angular momentum (l >= 5) functions before dumping molden file. If molden function raised error message `RuntimeError l=5 is not supported`, you can use this function to format orbitals.

Note the formated orbitals may have normalization problem. Some visualization tool will complain about the orbital normalization error.

Examples:

```
>>> mol1, orb1 = remove_high_l(mol, mf.mo_coeff)
>>> molden.from_mo(mol1, outputfile, orb1)
```

1.18.3 GAMESS WFN

GAMESS WFN File format

1.18.4 Cubegen

pyscf.tools.cubegen.**density**(*mol*, *outfile*, *dm*, *nx=80*, *ny=80*, *nz=80*)
Calculates electron density.

Args: mol (Mole): Molecule to calculate the electron density for. outfile (str): Name of Cube file to be written. dm (str): Density matrix of molecule. nx (int): Number of grid point divisions in x direction.

Note this is function of the molecule's size; a larger molecule will have a coarser representation than a smaller one for the same value.

ny (int): Number of grid point divisions in y direction. nz (int): Number of grid point divisions in z direction.

pyscf.tools.cubegen.**mep**(*mol*, *outfile*, *dm*, *nx=80*, *ny=80*, *nz=80*)
Calculates the molecular electrostatic potential (MEP).

Args: mol (Mole): Molecule to calculate the MEP for. outfile (str): Name of Cube file to be written. dm (str): Density matrix of molecule. nx (int): Number of grid point divisions in x direction.

Note this is function of the molecule's size; a larger molecule will have a coarser representation than a smaller one for the same value.

ny (int): Number of grid point divisions in y direction. nz (int): Number of grid point divisions in z direction.

1.18.5 Print Matrix

pyscf.tools.dump_mat.**dump_mo**(*mol*, *c*, *label=None*, *ncol=5*, *digits=5*, *start=1*)
Format print for orbitals

Args:

stdout [file object] eg sys.stdout, or stdout = open('/path/to/file') or mol.stdout if mol is an object initialized from `gto.Mole`

c [numpy.ndarray] Orbitals, each column is an orbital

Kwargs:

label [list of strings] Row labels (default is AO labels)

Examples:

```
>>> from pyscf import gto
>>> mol = gto.M(atom='C 0 0 0')
>>> mo = numpy.eye(mol.nao_nr())
>>> dump_mo(mol, mo)
             #0     #1     #2     #3     #4     #5     #6     #7     #8
0  C 1s      1.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
0  C 2s      0.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
0  C 3s      0.00   0.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00
0  C 2px     0.00   0.00   0.00   1.00   0.00   0.00   0.00   0.00   0.00
0  C 2py     0.00   0.00   0.00   0.00   1.00   0.00   0.00   0.00   0.00
0  C 2pz     0.00   0.00   0.00   0.00   0.00   1.00   0.00   0.00   0.00
0  C 3px     0.00   0.00   0.00   0.00   0.00   0.00   1.00   0.00   0.00
0  C 3py     0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00   0.00
0  C 3pz     0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00
```

pyscf.tools.dump_mat.**dump_rec**(*stdout*, *c*, *label=None*, *label2=None*, *ncol=5*, *digits=5*, *start=0*)

Print an array in rectangular format

Args:

stdout [file object] eg sys.stdout, or stdout = open('/path/to/file') or mol.stdout if mol is an object initialized from `gto.Mole`

c [numpy.ndarray] coefficients

Kwargs:

label [list of strings] Row labels (default is 1,2,3,4,...)

label2 [list of strings] Col labels (default is 1,2,3,4,...)

ncol [int] Number of columns in the format output (default 5)

digits [int] Number of digits of precision for floating point output (default 5)

start [int] The number to start to count the index (default 0)

Examples:

```
>>> import sys, numpy
>>> dm = numpy.eye(3)
>>> dump_rec(sys.stdout, dm)
        #0        #1        #2
0       1.00000   0.00000   0.00000
1       0.00000   1.00000   0.00000
2       0.00000   0.00000   1.00000
>>> from pyscf import gto
>>> mol = gto.M(atom='C 0 0 0')
>>> dm = numpy.eye(mol.nao_nr())
>>> dump_rec(sys.stdout, dm, label=mol.ao_labels(), ncol=9, digits=2)
             #0     #1     #2     #3     #4     #5     #6     #7     #8
0  C 1s      1.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
0  C 2s      0.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00
0  C 3s      0.00   0.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00
0  C 2px     0.00   0.00   0.00   1.00   0.00   0.00   0.00   0.00   0.00
0  C 2py     0.00   0.00   0.00   0.00   1.00   0.00   0.00   0.00   0.00
0  C 2pz     0.00   0.00   0.00   0.00   0.00   1.00   0.00   0.00   0.00
```

```
0  C 3px    0.00   0.00   0.00   0.00   0.00   0.00   1.00   0.00   0.00
0  C 3py    0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00   0.00
0  C 3pz    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00
```

pyscf.tools.dump_mat.**dump_tri**(*stdout*, *c*, *label=None*, *ncol=5*, *digits=5*, *start=0*)

Format print for the lower triangular part of an array

Args:

stdout [file object] eg sys.stdout, or stdout = open('/path/to/file') or mol.stdout if mol is an object initialized from `gto.Mole`

c [numpy.ndarray] coefficients

Kwargs:

label [list of strings] Row labels (default is 1,2,3,4,...)

ncol [int] Number of columns in the format output (default 5)

digits [int] Number of digits of precision for floating point output (default 5)

start [int] The number to start to count the index (default 0)

Examples:

```
>>> import sys, numpy
>>> dm = numpy.eye(3)
>>> dump_tri(sys.stdout, dm)
        #0        #1        #2
0       1.00000
1       0.00000   1.00000
2       0.00000   0.00000   1.00000
>>> from pyscf import gto
>>> mol = gto.M(atom='C 0 0 0')
>>> dm = numpy.eye(mol.nao_nr())
>>> dump_tri(sys.stdout, dm, label=mol.ao_labels(), ncol=9, digits=2)
            #0     #1     #2     #3     #4     #5     #6     #7     #8
0  C 1s     1.00
0  C 2s     0.00   1.00
0  C 3s     0.00   0.00   1.00
0  C 2px    0.00   0.00   0.00   1.00
0  C 2py    0.00   0.00   0.00   0.00   1.00
0  C 2pz    0.00   0.00   0.00   0.00   0.00   1.00
0  C 3px    0.00   0.00   0.00   0.00   0.00   0.00   1.00
0  C 3py    0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00
0  C 3pz    0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   1.00
```

1.19 grad — Analytical nuclear gradients

1.19.1 Analytical nuclear gradients

Simple usage:

```
>>> from pyscf import gto, scf, grad
>>> mol = gto.M(atom='N 0 0 0; N 0 0 1', basis='ccpvdz')
>>> mf = scf.RHF(mol).run()
>>> grad.RHF(mf).kernel()
```

1.20 hessian — Analytical nuclear Hessian

1.21 pbc — Periodic boundary conditions

The pbc module provides electronic structure implementations with periodic boundary conditions based on periodic Gaussian basis functions. The PBC implementation supports both all-electron and pseudopotential descriptions.

In PySCF, the PBC implementation has a tight relation to the molecular implementation. The module names, function names, and layouts of the PBC code are the same as (or as close as possible to) those of the molecular code. The PBC code supports the use (and mixing) of basis sets, pseudopotentials, and effective core potentials developed accross the materials science and quantum chemistry communites, offering great flexibility. Moreover, many post-mean-field methods defined in the molecular code can be seamlessly mixed with PBC calculations performed at the gamma point. For example, one can perform a gamma-point Hartree-Fock calculation in a supercell, followed by a CCSD(T) calculation, which is implemented in the molecular code.

In the PBC k-point calculations, we make small changes to the gamma-point data structures and export KHF and KDFT methods. On top of these KSCF methods, we have implemented k-point CCSD and k-point EOM-CCSD methods. Other post-mean-field methods can be analogously written to explicitly enforce translational symmetry through k-point sampling.

The list of modules described in this chapter is:

1.21.1 pbc.gto — Crystal cell structure

This module provides functions to setup the basic information of a PBC calculation. The pyscf.pbc.gto module is analogous to the basic molecular pyscf.gto module. The Cell class for crystal structure unit cells is defined in this module and is analogous to the basic molecular Mole class. Among other details, the basis set and pseudopotentials are parsed in this module.

Cell class

The *Cell* class is defined as an extension of the molecular *pyscf.gto.mole.Mole* class. The Cell object offers much of the same functionality as the Mole object. For example, one can use the Cell object to access the atomic structure, basis functions, pseudopotentials, and certain analytical periodic integrals.

Similar to the input in a molecular calculation, one first creates a Cell object. After assigning the crystal parameters, one calls build() to fully initialize the Cell object. A shortcut function M() is available at the module level to simplify the input.

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

import numpy
import pyscf.lib
from pyscf.pbc import gto

#
# Simliar to the initialization of "Mole" object, here we need create a "Cell"
# object for periodic boundary systems.
#
cell = gto.Cell()
cell.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'''
cell.basis = 'gth-szv'
cell.pseudo = 'gth-pade'
#
# Note the extra attribute ".a" in the "cell" initialization.
# .a is a matrix for lattice vectors.  Each row of .a is a primitive vector.
#
cell.a = numpy.eye(3)*3.5668
cell.build()

#
# pbc.gto module provided a shortcut initialization function "gto.M", like the
# one of finite size problem
#
cell = gto.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 = 'gth-szv',
    pseudo = 'gth-pade',
    a = numpy.eye(3)*3.5668)
```

Beyond the basic parameters `atom` and `basis`, one needs to set the unit cell lattice vectors `a` (a 3x3 array, where each row is a real-space primitive vector) and the numbers of grid points in the FFT-mesh in each positive direction `gs` (a length-3 list or 1x3 array); the total number of grid points is 2 `gs` +1.

In certain cases, it is convenient to choose the FFT-mesh density based on the kinetic energy cutoff. The `Cell` class offers an alternative attribute `ke_cutoff` that can be used to set the FFT-mesh. If `ke_cutoff` is set and `gs` is `None`, the `Cell` initialization function will convert the `ke_cutoff` to the equivalent FFT-mesh according to the relation $\mathbf{g} = \frac{\sqrt{2E_{\rm cut}}}{2\pi}\mathbf{a}^T$ and will overwrite the `gs` attribute.

Many PBC calculations are best performed using pseudopotentials, which are set via the `pseudo` attribute. Pseudopotentials alleviate the need for impractically dense FFT-meshes, although they represent a potentially uncontrolled source of error. See *Pseudo potential* for further details and a list of available pseudopotentials.

The input parameters `.a` and `.pseudo` are immutable in the `Cell` object. We emphasize that the input format might be different from the internal format used by PySCF. Similar to the convention in `Mole`, an internal Python data layer is created to hold the formatted `.a` and `.pseudo` parameters used as input.

_pseudo The internal format to hold PBC pseudo potential parameters. It is represented with nested Python lists only.

Nuclear-nuclear interaction energies are evaluated by means of Ewald summation, which depends on three parameters: the truncation radius for real-space lattice sums `rcut`, the Gaussian model charge `ew_eta`, and the energy cutoff `ew_cut`. Although they can be set manually, these parameters are by default chosen automatically according to the attribute `precision`, which likewise can be set manually or left to its default value.

Besides the methods and parameters provided by `Mole` class (see Chapter *gto — Molecular structure and GTO basis*), there are some parameters frequently used in the code to access the information of the crystal.

kpts The scaled or absolute k-points (nkpts x 3 array). This variable is not held as an attribute in `Cell` object;

instead, the `Cell` object provides functions to generate the k-points and convert the k-points between the scaled (fractional) value and absolute value:

```
# Generate k-points
n_kpts_each_direction = [2,2,2]
abs_kpts = cell.make_kpts(n_kpts_each_direction)

# Convert k-points between two convention, the scaled and the absoulte values
scaled_kpts = cell.get_scaled_kpts(abs_kpts)
abs_kpts = cell.get_abs_kpts(scaled_kpts)
```

Gv The (N x 3) array of plane waves associated to `gs`. `gs` defines the number of FFT grids in each direction. `Cell.Gv()` or `get_Gv()` convert the FFT-mesh to the plane waves. `Gv` are the the plane wave bases of 3D-FFT transformation. Given `gs = [nx,ny,nz]`, the number of vectors in `Gv` is `(2*nx+1)*(2*ny+1)*(2*nz+1)`.

vol `Cell.vol` gives the volume of the unit cell (in atomic unit).

reciprocal_vectors A 3x3 array. Each row is a reciprocal space primitive vector.

energy_nuc Similar to the `energy_nuc()` provided by `Mole` class, this function also return the energy associated to the nuclear repulsion. The nuclear repulsion energy is computed with Ewald summation technique. The background contribution is removed from the nuclear repulsion energy otherwise this term is divergent.

pbc_intor PBC analytic integral driver. It allows user to compute the PBC integral array in bulk, for given integral descriptor `intor` (see also `Mole.intor()` function *moleintor*). In the `Cell` object, we didn't overload the `intor()` method. So one can access both the periodic integrals and free-boundary integrals within the `Cell` object. It allows you to input the cell object into the molecule program to run the free-boundary calculation (see *Connection to Mole class*).

Note: `pbc_intor()` does not support Coulomb type integrals. Calling pbc_intor with Coulomb type integral descriptor such as `cint1e_nuc_sph` leads to divergent integrals. The Coulomb type PBC integrals should be evaluated with density fitting technique (see Chapter *pbc.df — PBC denisty fitting*).

Attributes and methods

class `pyscf.pbc.gto.`**`Cell`**`(`***kwargs*`)`

A Cell object holds the basic information of a crystal.

Attributes:

a [(3,3) ndarray] Lattice primitive vectors. Each row represents a lattice vector Reciprocal lattice vectors are given by b1,b2,b3 = 2 pi inv(a).T

gs [(3,) list of ints] The number of *positive* G-vectors along each direction. The default value is estimated based on `precision`

pseudo [dict or str] To define pseudopotential.

precision [float] To control Ewald sums and lattice sums accuracy

rcut [float] Cutoff radius (unit Bohr) in lattice summation. The default value is estimated based on the required `precision`.

ke_cutoff [float] If set, defines a spherical cutoff of planewaves, with .5 * G**2 < ke_cutoff The default value is estimated based on `precision`

dimension [int] Default is 3

** Following attributes (for experts) are automatically generated. **

ew_eta, ew_cut [float] The Ewald 'eta' and 'cut' parameters. See `get_ewald_params()`

(See other attributes in `Mole`)

Examples:

```
>>> mol = Mole(atom='H^2 0 0 0; H 0 0 1.1', basis='sto3g')
>>> cl = Cell()
>>> cl.build(a='3 0 0; 0 3 0; 0 0 3', gs=[8,8,8], atom='C 1 1 1', basis='sto3g')
>>> print(cl.atom_symbol(0))
C
```

bas_rcut (*cell*, *bas_id*, *precision=1e-08*)

Estimate the largest distance between the function and its image to reach the precision in overlap

precision ~ int g(r-0) g(r-R)

build (*dump_input=True*, *parse_arg=True*, *a=None*, *gs=None*, *ke_cutoff=None*, *precision=None*, *nimgs=None*, *ew_eta=None*, *ew_cut=None*, *pseudo=None*, *basis=None*, *h=None*, *dimension=None*, *rcut=None*, *ecp=None*, **args*, ***kwargs*)

Setup Mole molecule and Cell and initialize some control parameters. Whenever you change the value of the attributes of `Cell`, you need call this function to refresh the internal data of Cell.

Kwargs:

a [(3,3) ndarray] The real-space unit cell lattice vectors. Each row represents a lattice vector.

gs [(3,) ndarray of ints] The number of *positive* G-vectors along each direction.

pseudo [dict or str] To define pseudopotential. If given, overwrite `Cell.pseudo`

dumps (*cell*)

Serialize Cell object to a JSON formatted str.

energy_nuc (*cell*, *ew_eta=None*, *ew_cut=None*)

Perform real (R) and reciprocal (G) space Ewald sum for the energy.

Formulation of Martin, App. F2.

Returns:

float The Ewald energy consisting of overlap, self, and G-space sum.

See Also: pyscf.pbc.gto.get_ewald_params

ewald (*cell*, *ew_eta=None*, *ew_cut=None*)

Perform real (R) and reciprocal (G) space Ewald sum for the energy.

Formulation of Martin, App. F2.

Returns:

float The Ewald energy consisting of overlap, self, and G-space sum.

See Also: pyscf.pbc.gto.get_ewald_params

format_basis (*basis_tab*)

Convert the input `Cell.basis` to the internal data format:

```
{ atom: (l, kappa, ((-exp, c_1, c_2, ..), nprim, nctr, ptr-exps, ptr-
↪contraction-coeff)), ... }
```

Args:

basis_tab [dict] Similar to `Cell.basis`, it **cannot** be a str

Returns: Formated `basis`

Examples:

```
>>> pbc.format_basis({'H':'gth-szv'})
{'H': [[0,
    (8.3744350009, -0.0283380461),
    (1.8058681460, -0.1333810052),
    (0.4852528328, -0.3995676063),
    (0.1658236932, -0.5531027541)]]}
```

format_pseudo(*pseudo_tab*)

Convert the input `Cell.pseudo` (dict) to the internal data format:

```
{ atom: ( (nelec_s, nele_p, nelec_d, ...),
         rloc, nexp, (cexp_1, cexp_2, ..., cexp_nexp),
         nproj_types,
         (r1, nproj1, ( (hproj1[1,1], hproj1[1,2], ..., hproj1[1,nproj1]),
                        (hproj1[2,1], hproj1[2,2], ..., hproj1[2,nproj1]),
                        ...
                        (hproj1[nproj1,1], hproj1[nproj1,2], ...        ) )),
         (r2, nproj2, ( (hproj2[1,1], hproj2[1,2], ..., hproj2[1,nproj1]),
         ... ) )
         )
 ... }
```

Args:

pseudo_tab [dict] Similar to `Cell.pseudo` (a dict), it **cannot** be a str

Returns: Formatted `pseudo`

Examples:

```
>>> pbc.format_pseudo({'H':'gth-blyp', 'He': 'gth-pade'})
{'H': [[1],
    0.2, 2, [-4.19596147, 0.73049821], 0],
 'He': [[2],
    0.2, 2, [-9.1120234, 1.69836797], 0]}
```

from_ase(*ase_atom*)

Update cell based on given ase atom object

Examples:

```
>>> from ase.lattice import bulk
>>> cell.from_ase(bulk('C', 'diamond', a=LATTICE_CONST))
```

gen_uniform_grids(*cell*, *gs=None*)

Generate a uniform real-space grid consistent w/ samp thm; see MH (3.19).

Args: cell : instance of `Cell`

Returns:

coords [(ngx*ngy*ngz, 3) ndarray] The real-space grid point coordinates.

get_Gv(*cell*, *gs=None*)
Calculate three-dimensional G-vectors for the cell; see MH (3.8).

Indices along each direction go as [0...cell.gs, -cell.gs...-1] to follow FFT convention. Note that, for each direction, ngs = 2*cell.gs+1.

Args: cell : instance of `Cell`

Returns:

Gv [(ngs, 3) ndarray of floats] The array of G-vectors.

get_Gv_weights(*cell*, *gs=None*)
Calculate G-vectors and weights.

Returns:

Gv [(ngs, 3) ndarray of floats] The array of G-vectors.

get_SI(*cell*, *Gv=None*)
Calculate the structure factor for all atoms; see MH (3.34).

Args: cell : instance of `Cell`

Gv [(N,3) array] G vectors

Returns:

SI [(natm, ngs) ndarray, dtype=np.complex128] The structure factor for each atom at each G-vector.

get_abs_kpts(*scaled_kpts*)
Get absolute k-points (in 1/Bohr), given "scaled" k-points in fractions of lattice vectors.

Args: scaled_kpts : (nkpts, 3) ndarray of floats

Returns: abs_kpts : (nkpts, 3) ndarray of floats

get_bounding_sphere(*cell*, *rcut*)
Finds all the lattice points within a sphere of radius rcut.

Defines a parallelipiped given by -N_x <= n_x <= N_x, with x in [1,3] See Martin p. 85

Args:

rcut [number] real space cut-off for interaction

Returns: cut : ndarray of 3 ints defining N_x

get_ewald_params(*cell*, *precision=1e-08*, *gs=None*)
Choose a reasonable value of Ewald 'eta' and 'cut' parameters.

Choice is based on largest G vector and desired relative precision.

The relative error in the G-space sum is given by

precision ~ 4pi Gmax^2 e^{(-Gmax^2)/(4 eta^2)}

which determines eta. Then, real-space cutoff is determined by (exp. factors only)

precision ~ erfc(eta*rcut) / rcut ~ e^{(-eta**2 rcut*2)}

Returns:

ew_eta, ew_cut [float] The Ewald 'eta' and 'cut' parameters.

get_lattice_Ls(*cell*, *nimgs=None*, *rcut=None*, *dimension=None*)
Get the (Cartesian, unitful) lattice translation vectors for nearby images. The translation vectors can be used for the lattice summation.

get_nimgs(*cell*, *precision=None*)
Choose number of basis function images in lattice sums to include for given precision in overlap, using

precision ~ int r^l e^{-alpha r^2} (r-rcut)^l e^{-alpha (r-rcut)^2} ~ (rcut^2/(2alpha))^l e^{alpha/2 rcut^2}

where alpha is the smallest exponent in the basis. Note that assumes an isolated exponent in the middle of the box, so it adds one additional lattice vector to be safe.

get_scaled_kpts(*abs_kpts*)
Get scaled k-points, given absolute k-points in 1/Bohr.

Args: abs_kpts : (nkpts, 3) ndarray of floats

Returns: scaled_kpts : (nkpts, 3) ndarray of floats

has_ecp()
Whether pesudo potential is used in the system.

kernel(*dump_input=True*, *parse_arg=True*, *a=None*, *gs=None*, *ke_cutoff=None*, *precision=None*, *nimgs=None*, *ew_eta=None*, *ew_cut=None*, *pseudo=None*, *basis=None*, *h=None*, *dimension=None*, *rcut=None*, *ecp=None*, **args*, ***kwargs*)
Setup Mole molecule and Cell and initialize some control parameters. Whenever you change the value of the attributes of `Cell`, you need call this function to refresh the internal data of Cell.

Kwargs:

a [(3,3) ndarray] The real-space unit cell lattice vectors. Each row represents a lattice vector.

gs [(3,) ndarray of ints] The number of *positive* G-vectors along each direction.

pseudo [dict or str] To define pseudopotential. If given, overwrite `Cell.pseudo`

lattice_vectors()
Convert the primitive lattice vectors.

Return 3x3 array in which each row represents one direction of the lattice vectors (unit in Bohr)

loads(*molstr*)
Deserialize a str containing a JSON document to a Cell object.

make_kpts(*cell*, *nks*, *wrap_around=False*, *with_gamma_point=True*)
Given number of kpoints along x,y,z , generate kpoints

Args: nks : (3,) ndarray

Kwargs:

wrap_around [bool] To ensure all kpts are in first Brillouin zone.

with_gamma_point [bool] Whether to shift Monkhorst-pack grid to include gamma-point.

Returns: kpts in absolute value (unit 1/Bohr). Gamma point is placed at the first place in the k-points list

Examples:

```
>>> cell.make_kpts((4,4,4))
```

pack(*cell*)
Pack the input args of `Cell` to a dict, which can be serialized with `pickle`

pbc_intor(*intor*, *comp=1*, *hermi=0*, *kpts=None*, *kpt=None*)
One-electron integrals with PBC. See also Mole.intor

reciprocal_vectors(*norm_to=6.283185307179586*)

$$\mathbf{b_1} = 2\pi \frac{\mathbf{a_2} \times \mathbf{a_3}}{\mathbf{a_1} \cdot (\mathbf{a_2} \times \mathbf{a_3})} \tag{1.4}$$

$$\mathbf{b_2} = 2\pi \frac{\mathbf{a_3} \times \mathbf{a_1}}{\mathbf{a_2} \cdot (\mathbf{a_3} \times \mathbf{a_1})} \tag{1.5}$$

$$\mathbf{b_3} = 2\pi \frac{\mathbf{a_1} \times \mathbf{a_2}}{\mathbf{a_3} \cdot (\mathbf{a_1} \times \mathbf{a_2})} \tag{1.6}$$

to_mol()
Return a Mole object using the same atoms and basis functions as the Cell object.

unpack(*moldic*)
Convert the packed dict to a `Cell` object, to generate the input arguments for `Cell` object.

Connection to `Mole` class

`Cell` class is compatible with the molecule *`pyscf.gto.mole.Mole`* class. They shared most data structure and methods. It gives the freedom to mix the finite size calculation and the PBC calculation. If you feed the cell object to molecule module/functions, the molecule program will not check whether the given `Mole` object is the true `Mole` or not. It simply treats the `Cell` object as the `Mole` object and run the finite size calculations. Because the same module names were used in PBC program and molecule program, you should be careful with the imported modules since no error message will be raised if you by mistake input the `Cell` object into the molecule program.

Although we reserve the flexibility to mix the `Cell` and `Mole` objects in the same code, it should be noted that the serialization methods of the two objects are not completely compatible. When you dumps/loads the cell object in the molecule program, informations of the `Cell` object or the faked `Mole` object may be lost.

Serialization

`Cell` class has two set of functions to serialize Cell object in different formats.

- JSON format is the default serialization format used by *`pyscf.lib.chkfile`* module. It can be serialized by `Cell.dumps()` function and deserialized by `Cell.loads()` function.
- In the old version, `Mole.pack()` and `Mole.unpack()` functions are used to convert the `Mole` object to and from Python dict. The Python dict is then serialized by pickle module. This serialization method is not used anymore in the new PySCF code. To keep the backward compatibility, the two methods are defined in `Cell` class.

Basis set

The pbc module supports all-electron calculation. The all-electron basis sets developed by quantum chemistry community can be directly used in the pbc calculation. The `Cell` class supports to mix the QC all-electron basis and PBC basis in the same calculation.

```
#!/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]
O    S
    130.7093200              0.15432897
     23.8088610              0.53532814
      6.4436083              0.44463454
O    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)
```

Note: The default PBC Coulomb type integrals are computed using FFT transformation. If the all-electron basis are used, you might need very high energy cutoff to converge the integrals. It is recommended to use mixed density fitting technique (*pbc.df — PBC denisty fitting*) to handle the all-electron calculations.

Pseudo potential

Quantum chemistry community developed a wide range of pseudo potentials (which are called ECP, effective core potential) for heavy elements. ECP works quite successful in finite system. It has high flexibility to choose different core size and relevant basis sets to satisfy different requirements on accuracy, efficiency in different simulation scenario. Extending ECP to PBC code enriches the pseudo potential database. PySCF PBC program supports both the PBC conventional pseudo potential and ECP and the mix of the two kinds of potentials in the same calculation.

```
#!/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 potentail (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 single zeta␣
↪basis
                      '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,
    gs = [4]*3,
    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'})
```

1.21.2 pbc.scf — Mean-field with periodic boundary condition

This module is an analogy to molecular *pyscf.scf* module to handle mean-filed calculation with periodic boundary condition.

Gamma point and single k-point calculation

The usage of gamma point Hartree-Fock program is very close to that of the molecular program. In the PBC gamma point calculation, one needs intialize `Cell` object and the corresponding *pyscf.pbc.scf.hf.RHF* class:

```
from pyscf.pbc import gto, scf
cell = gto.M(
    atom = '''H     0.      0.      0.
              H     0.8917  0.8917  0.8917''',
    basis = 'sto3g',
    h = '''
    0       1.7834  1.7834
    1.7834  0       1.7834
    1.7834  1.7834  0     ''',
    gs = [10]*3,
    verbose = 4,
)
mf = scf.RHF(cell).run()
```

Comparing to the *pyscf.scf.hf.RHF* object for molecular calculation, the PBC-HF calculation with *pyscf.pbc.scf.hf.RHF* or *pyscf.pbc.scf.uhf.UHF* has three differences

- `psycf.pbc.scf.hf.RHF` is the single k-point PBC HF class. By default, it creates the gamma point calculation. You can change to other k-point by setting the `kpt` attribute:

  ```
  mf = scf.RHF(cell)
  mf.kpt = cell.get_abs_kpts([.25,.25,.25])  # convert from scaled kpts
  mf.kernel()
  ```

- The exchange integrals of the PBC Hartree-Fock method has slow convergence with respect to the number of k-points. Proper treatments for the divergent part of exchange integrals can effectively improve the convergence. Attribute `exxdive` is used to control the method to handle exchange divergent term. The default `exxdiv='ewald'` is favored in most scenario. However, if the molecular post-HF methods was mixed with the gamma point HF method (see *Mixing with molecular program for post-HF methods*, you might need set `exxdiv=None` to get consistent total energy (see *Exchange divergence treatment*).

- In the finite-size system, one can obtain right answer without considering the model to evaluate 2-electron integrals. But the integral scheme might need to be updated in the PBC calculations. The default integral scheme is accurate for pseudo-potential. In the all-electron calculation, you may need change the `with_df` attribute to mixed density fitting (MDF) method for better accuracy (see *with_df for density fitting*). Here is an example to update `with_df`

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

'''
Gamma point Hartree-Fock/DFT for all-electron calculation
```

```
The default FFT-based 2-electron integrals may not be accurate enough for
all-electron calculation.  It's recommended to use MDF (mixed density fitting)
technique to improve the accuracy.

See also
examples/df/00-with_df.py
examples/df/01-auxbasis.py
examples/df/40-precomupte_df_ints.py
'''

import numpy
from pyscf.pbc import gto, scf, dft

cell = gto.M(
    a = numpy.eye(3)*3.5668,
    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 = '6-31g',
    verbose = 4,
)

mf = scf.RHF(cell).density_fit()
mf.kernel()

# Mixed density fitting is another option for all-electron calculations
mf = scf.RHF(cell).mix_density_fit()
mf.with_df.gs = [5]*3  # Tune #PWs in MDF for performance/accuracy balance
mf.kernel()

# Or use even-tempered Gaussian basis as auxiliary fitting functions.
# The following auxbasis is generated based on the expression
#    alpha = a * 1.7^i   i = 0..N
# where a and N are determined by the smallest and largest exponents of AO basis.
import pyscf.df
auxbasis = pyscf.df.aug_etb(cell, beta=1.7)
mf = scf.RHF(cell).density_fit(auxbasis=auxbasis)
mf.kernel()

#
# Second order SCF solver can be used in the PBC SCF code the same way in the
# molecular calculation
#
mf = dft.RKS(cell).density_fit(auxbasis='weigend')
mf.xc = 'bp86'
mf = scf.newton(mf)
mf.kernel()

#
# The computational costs to initialize PBC DF object is high.  The density
# fitting integral tensor created in the initialization can be cached for
# future use.  See also examples/df/40-precomupte_df_ints.py
```

```
#
mf = dft.RKS(cell).density_fit(auxbasis='weigend')
mf.with_df._cderi_to_save = 'df_ints.h5'
mf.kernel()
#
# The DF integral tensor can be preloaded in an independent calculation.
#
mf = dft.RKS(cell).density_fit()
mf.with_df._cderi = 'df_ints.h5'
mf.kernel()
```

Mixing with molecular program for post-HF methods

The gamma point HF code adopts the same code structure, the function and method names and the arguments' convention as the molecule SCF code. This desgin allows one mixing PBC HF object with the existed molecular post-HF code for PBC electron correlation problems. A typical molecular post-HF calculation starts from the finite-size HF method with the `Mole` object:

```
from pyscf import gto, scf
mol = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz')
mf = scf.RHF(mol).run()

from pyscf import cc
cc.CCSD(mf).run()
```

The PBC gamma point post-HF calculation requires the `Cell` object and PBC HF object:

```
from pyscf.pbc import gto, scf
cell = gto.M(atom='H 0 0 0; H 0 0 1', basis='ccpvdz',
             h=numpy.eye(3)*2, gs=[10,10,10])
mf = scf.RHF(cell).run()

from pyscf import cc
cc.CCSD(mf).run()
```

The differences are the the `mol` or `cell` object to create and the `scf` module to import. With the system-specific mean-field object, one can carray out various post-HF methods (MP2, Coupled cluster, CISD, TDHF, TDDFT, ...) using the same code for finite-size and extended systems. See *Mixing PBC and molecular modules* for more details of the interface between PBC and molecular modules.

k-point sampling

Newton solver

Smearing

Exchange divergence treatment

The PBC Hartree-Fock has slow convergence of exchange integral with respect to the number of k-points. In the single k-point calculation, Generally, exxdiv leads to a shift in the total energy and the spectrum of orbital energy. It should not affect the following correlation energy in the post-HF calculation. In practice, when the gamma-point calculation is mixed with molecular program eg the FCI solver, the exxdiv attribute may leads to inconsistency in the total energy.

`with_df` for density fitting

Placing the `with_df` attribute in SCF object to get the compatibility to molecule DF-SCF methods.

Stability analysis

Program reference

Hartree-Fock for periodic systems at a single k-point

See Also: pyscf.pbc.scf.khf.py : Hartree-Fock for periodic systems with k-point sampling

`pyscf.pbc.scf.hf.`**`RHF`**
alias of *SCF*

class `pyscf.pbc.scf.hf.`**`SCF`**(*cell*, *kpt=array([0., 0., 0.])*, *exxdiv='ewald'*)
SCF class adapted for PBCs.

Attributes:

kpt [(3,) ndarray] The AO k-point in Cartesian coordinates, in units of 1/Bohr.

exxdiv [str] Exchange divergence treatment, can be one of

None : ignore G=0 contribution in exchange integral
'ewald' : Ewald summation for G=0 in exchange integral

with_df [density fitting object] Default is the FFT based DF model. For all-electron calculation, MDF model is favored for better accuracy. See also `pyscf.pbc.df`.

direct_scf [bool] When this flag is set to true, the J/K matrices will be computed directly through the underlying with_df methods. Otherwise, depending the available memory, the 4-index integrals may be cached and J/K matrices are computed based on the 4-index integrals.

`get_bands`(*mf*, *kpts_band*, *cell=None*, *dm=None*, *kpt=None*)
Get energy bands at the given (arbitrary) 'band' k-points.

Returns:

mo_energy [(nmo,) ndarray or a list of (nmo,) ndarray] Bands energies E_n(k)

mo_coeff [(nao, nmo) ndarray or a list of (nao,nmo) ndarray] Band orbitals psi_n(k)

`get_j`(*cell=None*, *dm=None*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Compute J matrix for the given density matrix.

`get_jk`(*cell=None*, *dm=None*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Get Coulomb (J) and exchange (K) following `scf.hf.RHF.get_jk_()`.

Note the incore version, which initializes an _eri array in memory.

`get_jk_incore`(*cell=None*, *dm=None*, *hermi=1*, *verbose=5*, *kpt=None*)
Get Coulomb (J) and exchange (K) following `scf.hf.RHF.get_jk_()`.

Incore version of Coulomb and exchange build only. Currently RHF always uses PBC AO integrals (unlike RKS), since exchange is currently computed by building PBC AO integrals.

`get_k`(*cell=None*, *dm=None*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Compute K matrix for the given density matrix.

get_veff(*cell=None*, *dm=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Hartree-Fock potential matrix for the given density matrix. See `scf.hf.get_veff()` and `scf.hf.RHF.get_veff()`

pyscf.pbc.scf.hf.**dot_eri_dm**(*eri*, *dm*, *hermi=0*)
Compute J, K matrices in terms of the given 2-electron integrals and density matrix. eri or dm can be complex.

Args:

eri [ndarray] complex integral array with N^4 elements (N is the number of orbitals)

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian

0 : no hermitian or symmetric
1 : hermitian
2 : anti-hermitian

Returns: Depending on the given dm, the function returns one J and one K matrix, or a list of J matrices and a list of K matrices, corresponding to the input density matrices.

pyscf.pbc.scf.hf.**get_bands**(*mf*, *kpts_band*, *cell=None*, *dm=None*, *kpt=None*)
Get energy bands at the given (arbitrary) 'band' k-points.

Returns:

mo_energy [(nmo,) ndarray or a list of (nmo,) ndarray] Bands energies E_n(k)

mo_coeff [(nao, nmo) ndarray or a list of (nao,nmo) ndarray] Band orbitals psi_n(k)

pyscf.pbc.scf.hf.**get_hcore**(*cell*, *kpt=array([0., 0., 0.])*)
Get the core Hamiltonian AO matrix.

pyscf.pbc.scf.hf.**get_j**(*cell*, *dm*, *hermi=1*, *vhfopt=None*, *kpt=array([0., 0., 0.])*, *kpts_band=None*)
Get the Coulomb (J) AO matrix for the given density matrix.

Args:

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian | 0 : no hermitian or symmetric | 1 : hermitian | 2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

kpt [(3,) ndarray] The "inner" dummy k-point at which the DM was evaluated (or sampled).

kpts_band [(3,) ndarray or (*,3) ndarray] An arbitrary "band" k-point at which J is evaluated.

Returns: The function returns one J matrix, corresponding to the input density matrix (both order and shape).

pyscf.pbc.scf.hf.**get_jk**(*mf*, *cell*, *dm*, *hermi=1*, *vhfopt=None*, *kpt=array([0., 0., 0.])*, *kpts_band=None*)
Get the Coulomb (J) and exchange (K) AO matrices for the given density matrix.

Args:

dm [ndarray or list of ndarrays] A density matrix or a list of density matrices

Kwargs:

hermi [int] Whether J, K matrix is hermitian | 0 : no hermitian or symmetric | 1 : hermitian | 2 : anti-hermitian

vhfopt : A class which holds precomputed quantities to optimize the computation of J, K matrices

kpt [(3,) ndarray] The "inner" dummy k-point at which the DM was evaluated (or sampled).

kpts_band [(3,) ndarray or (*,3) ndarray] An arbitrary "band" k-point at which J and K are evaluated.

Returns: The function returns one J and one K matrix, corresponding to the input density matrix (both order and shape).

pyscf.pbc.scf.hf.**get_nuc**(*cell*, *kpt=array([0., 0., 0.])*)
Get the bare periodic nuc-el AO matrix, with G=0 removed.

See Martin (12.16)-(12.21).

pyscf.pbc.scf.hf.**get_ovlp**(*cell*, *kpt=array([0., 0., 0.])*)
Get the overlap AO matrix.

pyscf.pbc.scf.hf.**get_t**(*cell*, *kpt=array([0., 0., 0.])*)
Get the kinetic energy AO matrix.

pyscf.pbc.scf.hf.**init_guess_by_chkfile**(*cell*, *chkfile_name*, *project=True*, *kpt=None*)
Read the HF results from checkpoint file, then project it to the basis defined by `cell`

Returns: Density matrix, (nao,nao) ndarray

Unrestricted Hartree-Fock for periodic systems at a single k-point

See Also: pyscf.pbc.scf.khf.py : Hartree-Fock for periodic systems with k-point sampling

class pyscf.pbc.scf.uhf.**UHF**(*cell*, *kpt=array([0., 0., 0.])*, *exxdiv='ewald'*)
UHF class for PBCs.

energy_tot(*mf*, *dm=None*, *h1e=None*, *vhf=None*)
Total Hartree-Fock energy, electronic part plus nuclear repulstion See `scf.hf.energy_elec()` for the electron part

get_bands(*kpts_band*, *cell=None*, *dm=None*, *kpt=None*)
Get energy bands at the given (arbitrary) 'band' k-points.

Returns:

mo_energy [(nmo,) ndarray or a list of (nmo,) ndarray] Bands energies E_n(k)

mo_coeff [(nao, nmo) ndarray or a list of (nao,nmo) ndarray] Band orbitals psi_n(k)

get_j(*cell=None*, *dm=None*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Compute J matrix for the given density matrix.

get_jk(*cell=None*, *dm=None*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Get Coulomb (J) and exchange (K) following `scf.hf.RHF.get_jk_()`.

Note the incore version, which initializes an _eri array in memory.

get_jk_incore(*cell=None*, *dm=None*, *hermi=1*, *verbose=5*, *kpt=None*)
Get Coulomb (J) and exchange (K) following `scf.hf.RHF.get_jk_()`.

Incore version of Coulomb and exchange build only. Currently RHF always uses PBC AO integrals (unlike RKS), since exchange is currently computed by building PBC AO integrals.

get_k(*cell=None*, *dm=None*, *hermi=1*, *kpt=None*, *kpts_band=None*)
Compute K matrix for the given density matrix.

pyscf.pbc.scf.uhf.**init_guess_by_chkfile**(*cell*, *chkfile_name*, *project=True*, *kpt=None*)
Read the HF results from checkpoint file, then project it to the basis defined by `cell`

Returns: Density matrix, (nao,nao) ndarray

Hartree-Fock for periodic systems with k-point sampling

See Also: hf.py : Hartree-Fock for periodic systems at a single k-point

pyscf.pbc.scf.khf.**KRHF**
alias of *KSCF*

class pyscf.pbc.scf.khf.**KSCF**(*cell*, *kpts=array([[0., 0., 0.]])*, *exxdiv='ewald'*)
SCF class with k-point sampling.

Compared to molecular SCF, some members such as mo_coeff, mo_occ now have an additional first dimension for the k-points, e.g. mo_coeff is (nkpts, nao, nao) ndarray

Attributes:

kpts [(nks,3) ndarray] The sampling k-points in Cartesian coordinates, in units of 1/Bohr.

energy_elec(*mf*, *dm_kpts=None*, *h1e_kpts=None*, *vhf_kpts=None*)
Following pyscf.scf.hf.energy_elec()

get_bands(*kpts_band*, *cell=None*, *dm_kpts=None*, *kpts=None*)
Get energy bands at the given (arbitrary) 'band' k-points.

Returns:

mo_energy [(nmo,) ndarray or a list of (nmo,) ndarray] Bands energies E_n(k)

mo_coeff [(nao, nmo) ndarray or a list of (nao,nmo) ndarray] Band orbitals psi_n(k)

get_fermi(*mf*, *mo_energy_kpts=None*, *mo_occ_kpts=None*)
Fermi level

get_grad(*mo_coeff_kpts*, *mo_occ_kpts*, *fock=None*)
returns 1D array of gradients, like non K-pt version note that occ and virt indices of different k pts now occur in sequential patches of the 1D array

get_occ(*mf*, *mo_energy_kpts=None*, *mo_coeff_kpts=None*)
Label the occupancies for each orbital for sampled k-points.

This is a k-point version of scf.hf.SCF.get_occ

get_ovlp(*mf*, *cell=None*, *kpts=None*)
Get the overlap AO matrices at sampled k-points.

Args: kpts : (nkpts, 3) ndarray

Returns: ovlp_kpts : (nkpts, nao, nao) ndarray

get_veff(*cell=None*, *dm_kpts=None*, *dm_last=0*, *vhf_last=0*, *hermi=1*, *kpts=None*, *kpts_band=None*)
Hartree-Fock potential matrix for the given density matrix. See `scf.hf.get_veff()` and `scf.hf.RHF.get_veff()`

pyscf.pbc.scf.khf.**analyze**(*mf*, *verbose=5*, ***kwargs*)
Analyze the given SCF object: print orbital energies, occupancies; print orbital coefficients; Mulliken population analysis; Dipole moment

pyscf.pbc.scf.khf.**energy_elec**(*mf*, *dm_kpts=None*, *h1e_kpts=None*, *vhf_kpts=None*)
Following pyscf.scf.hf.energy_elec()

pyscf.pbc.scf.khf.**get_fermi**(*mf*, *mo_energy_kpts=None*, *mo_occ_kpts=None*)
Fermi level

pyscf.pbc.scf.khf.**get_grad**(*mo_coeff_kpts*, *mo_occ_kpts*, *fock*)
returns 1D array of gradients, like non K-pt version note that occ and virt indices of different k pts now occur in sequential patches of the 1D array

pyscf.pbc.scf.khf.**get_hcore**(*mf*, *cell=None*, *kpts=None*)
Get the core Hamiltonian AO matrices at sampled k-points.

Args: kpts : (nkpts, 3) ndarray

Returns: hcore : (nkpts, nao, nao) ndarray

pyscf.pbc.scf.khf.**get_j**(*mf*, *cell*, *dm_kpts*, *kpts*, *kpts_band=None*)
Get the Coulomb (J) AO matrix at sampled k-points.

Args:

dm_kpts [(nkpts, nao, nao) ndarray or a list of (nkpts,nao,nao) ndarray] Density matrix at each k-point. If a list of k-point DMs, eg, UHF alpha and beta DM, the alpha and beta DMs are contracted separately.

Kwargs:

kpts_band [(k,3) ndarray] A list of arbitrary "band" k-points at which to evalute the matrix.

Returns: vj : (nkpts, nao, nao) ndarray or list of vj if the input dm_kpts is a list of DMs

pyscf.pbc.scf.khf.**get_jk**(*mf*, *cell*, *dm_kpts*, *kpts*, *kpts_band=None*)
Get the Coulomb (J) and exchange (K) AO matrices at sampled k-points.

Args:

dm_kpts [(nkpts, nao, nao) ndarray] Density matrix at each k-point

Kwargs:

kpts_band [(3,) ndarray] A list of arbitrary "band" k-point at which to evalute the matrix.

Returns: vj : (nkpts, nao, nao) ndarray vk : (nkpts, nao, nao) ndarray or list of vj and vk if the input dm_kpts is a list of DMs

pyscf.pbc.scf.khf.**get_occ**(*mf*, *mo_energy_kpts=None*, *mo_coeff_kpts=None*)
Label the occupancies for each orbital for sampled k-points.

This is a k-point version of scf.hf.SCF.get_occ

pyscf.pbc.scf.khf.**get_ovlp**(*mf*, *cell=None*, *kpts=None*)
Get the overlap AO matrices at sampled k-points.

Args: kpts : (nkpts, 3) ndarray

Returns: ovlp_kpts : (nkpts, nao, nao) ndarray

pyscf.pbc.scf.khf.**init_guess_by_chkfile**(*cell*, *chkfile_name*, *project=True*, *kpts=None*)
Read the KHF results from checkpoint file, then project it to the basis defined by `cell`

Returns: Density matrix, 3D ndarray

pyscf.pbc.scf.khf.**make_rdm1**(*mo_coeff_kpts*, *mo_occ_kpts*)
One particle density matrices for all k-points.

Returns: dm_kpts : (nkpts, nao, nao) ndarray

pyscf.pbc.scf.khf.**mulliken_meta**(*cell*, *dm_ao*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)
Mulliken population analysis, based on meta-Lowdin AOs.

Note this function only computes the Mulliken population for the gamma point density matrix.

Hartree-Fock for periodic systems with k-point sampling

See Also: hf.py : Hartree-Fock for periodic systems at a single k-point

class pyscf.pbc.scf.kuhf.**KUHF**(*cell*, *kpts=array([[0., 0., 0.]])*, *exxdiv='ewald'*)

UHF class with k-point sampling.

canonicalize(*mf*, *mo_coeff_kpts*, *mo_occ_kpts*, *fock=None*)

Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separatedly (without change occupancy).

energy_elec(*mf*, *dm_kpts=None*, *h1e_kpts=None*, *vhf_kpts=None*)

Following pyscf.scf.hf.energy_elec()

get_bands(*kpts_band*, *cell=None*, *dm_kpts=None*, *kpts=None*)

Get energy bands at the given (arbitrary) 'band' k-points.

Returns:

mo_energy [(nmo,) ndarray or a list of (nmo,) ndarray] Bands energies E_n(k)

mo_coeff [(nao, nmo) ndarray or a list of (nao,nmo) ndarray] Band orbitals psi_n(k)

get_occ(*mf*, *mo_energy_kpts=None*, *mo_coeff_kpts=None*)

Label the occupancies for each orbital for sampled k-points.

This is a k-point version of scf.hf.SCF.get_occ

get_ovlp(*mf*, *cell=None*, *kpts=None*)

Get the overlap AO matrices at sampled k-points.

Args: kpts : (nkpts, 3) ndarray

Returns: ovlp_kpts : (nkpts, nao, nao) ndarray

spin_square(*mo_coeff=None*, *s=None*)

Spin of the given UHF orbitals

$$S^2 = \frac{1}{2}(S_+S_- + S_-S_+) + S_z^2$$

where $S_+ = \sum_i S_{i+}$ is effective for all beta occupied orbitals; $S_- = \sum_i S_{i-}$ is effective for all alpha occupied orbitals.

1.**There are two possibilities for** S_+S_-

(a) same electron $S_+S_- = \sum_i s_{i+}s_{i-}$,

$$\sum_i \langle UHF|s_{i+}s_{i-}|UHF\rangle = \sum_{pq}\langle p|s_+s_-|q\rangle\gamma_{qp} = n_\alpha$$

2) different electrons $S_+S_- = \sum s_{i+}s_{j-}, (i \neq j)$. There are in total $n(n-1)$ terms. As a two-particle operator,

$$\langle S_+S_-\rangle = \langle ij|s_+s_-|ij\rangle - \langle ij|s_+s_-|ji\rangle = -\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle$$

2.**Similarly, for** S_-S_+

(a) same electron

$$\sum_i \langle s_{i-}s_{i+}\rangle = n_\beta$$

(a) different electrons

$$\langle S_- S_+ \rangle = -\langle i^\beta | j^\alpha \rangle \langle j^\alpha | i^\beta \rangle$$

2.**For** S_z^2

(a) same electron

$$\langle s_z^2 \rangle = \frac{1}{4}(n_\alpha + n_\beta)$$

(a) different electrons

$$\begin{aligned}
&\frac{1}{2}\sum_{ij}(\langle ij|2s_{z1}s_{z2}|ij\rangle - \langle ij|2s_{z1}s_{z2}|ji\rangle) \\
&= \frac{1}{4}(\langle i^\alpha|i^\alpha\rangle\langle j^\alpha|j^\alpha\rangle - \langle i^\alpha|i^\alpha\rangle\langle j^\beta|j^\beta\rangle - \langle i^\beta|i^\beta\rangle\langle j^\alpha|j^\alpha\rangle + \langle i^\beta|i^\beta\rangle\langle j^\beta|j^\beta\rangle) \\
&- \frac{1}{4}(\langle i^\alpha|j^\alpha\rangle\langle j^\alpha|i^\alpha\rangle + \langle i^\beta|j^\beta\rangle\langle j^\beta|i^\beta\rangle) \\
&= \frac{1}{4}(n_\alpha^2 - n_\alpha n_\beta - n_\beta n_\alpha + n_\beta^2) - \frac{1}{4}(n_\alpha + n_\beta) \\
&= \frac{1}{4}((n_\alpha - n_\beta)^2 - (n_\alpha + n_\beta))
\end{aligned}$$

In total

$$\langle S^2 \rangle = \frac{1}{2}(n_\alpha - \sum_{ij}\langle i^\alpha|j^\beta\rangle\langle j^\beta|i^\alpha\rangle + n_\beta - \sum_{ij}\langle i^\beta|j^\alpha\rangle\langle j^\alpha|i^\beta\rangle) + \frac{1}{4}(n_\alpha - n_\beta)^2$$

Args:

mo [a list of 2 ndarrays] Occupied alpha and occupied beta orbitals

Kwargs:

s [ndarray] AO overlap

Returns: A list of two floats. The first is the expectation value of S^2. The second is the corresponding 2S+1

Examples:

```
>>> mol = gto.M(atom='O 0 0 0; H 0 0 1; H 0 1 0', basis='ccpvdz', charge=1,␣
↪spin=1, verbose=0)
>>> mf = scf.UHF(mol)
>>> mf.kernel()
-75.623975516256706
>>> mo = (mf.mo_coeff[0][:,mf.mo_occ[0]>0], mf.mo_coeff[1][:,mf.mo_occ[1]>0])
>>> print('S^2 = %.7f, 2S+1 = %.7f' % spin_square(mo, mol.intor('int1e_ovlp_
↪sph')))
S^2 = 0.7570150, 2S+1 = 2.0070027
```

pyscf.pbc.scf.kuhf.**canonicalize**(*mf*, *mo_coeff_kpts*, *mo_occ_kpts*, *fock=None*)
Canonicalization diagonalizes the UHF Fock matrix within occupied, virtual subspaces separatedly (without change occupancy).

pyscf.pbc.scf.kuhf.**energy_elec**(*mf*, *dm_kpts=None*, *h1e_kpts=None*, *vhf_kpts=None*)
Following pyscf.scf.hf.energy_elec()

pyscf.pbc.scf.kuhf.**get_occ**(*mf*, *mo_energy_kpts=None*, *mo_coeff_kpts=None*)
Label the occupancies for each orbital for sampled k-points.

This is a k-point version of scf.hf.SCF.get_occ

pyscf.pbc.scf.kuhf.**init_guess_by_chkfile**(*cell*, *chkfile_name*, *project=True*, *kpts=None*)
Read the KHF results from checkpoint file, then project it to the basis defined by `cell`

Returns: Density matrix, 3D ndarray

pyscf.pbc.scf.kuhf.**make_rdm1**(*mo_coeff_kpts*, *mo_occ_kpts*)
Alpha and beta spin one particle density matrices for all k-points.

Returns: dm_kpts : (2, nkpts, nao, nao) ndarray

pyscf.pbc.scf.kuhf.**mulliken_meta**(*cell*, *dm_ao_kpts*, *verbose=5*, *pre_orth_method='ANO'*, *s=None*)
Mulliken population analysis, based on meta-Lowdin AOs.

Note this function only computes the Mulliken population for the gamma point density matrix.

1.21.3 pbc.dft

1.21.4 pbc.df — PBC denisty fitting

Introduction

The *pbc.df* module provides the fundamental functions to handle the density fitting (DF) integral tensors required by the gamma-point and k-point PBC calculations. There are four types of DF methods available for PBC systems. They are FFTDF (plane-wave density fitting with fast Fourier transformation), AFTDF (plane-wave density fitting with analytical Fourier transformation), GDF (Gaussian density fitting) and MDF (mixed density fitting). The Coulomb integrals and nuclear attraction integrals in the PBC calculations are all computed with DF technique. The default scheme is FFTDF.

The characters of these PBC DF methods are summarized in the following table

Subject	FFTDF	AFTDF	GDF	MDF
Initialization	No	No	Slow	Slow
HF Coulomb matrix (J)	Fast	Slow	Fast	Moderate
HF exchange matrix (K)	Slow	Slow	Fast	Moderate
Building ERIs	Slow	Slow	Fast	Moderate
All-electron calculation	Huge error	Large error	Accurate	Most accurate
Low-dimension system	N/A	0D,1D,2D	0D,1D,2D	0D,1D,2D

FFTDF — FFT-based density fitting

FFTDF represents the method to compute electron repulsion integrals in reciprocal space with the Fourier transformed Coulomb kernel

$$(ij|kl) = \sum_G \rho_{ij}(\mathbf{G}) \frac{4\pi}{G^2} \rho_{kl}(-\mathbf{G})$$

$\mathbf{G}$ is the plane wave vector. $\rho_{ij}(\mathbf{G})$ is the Fourier transformed orbital pair

$$\rho_{ij}(\mathbf{G}) = \sum_r e^{-\mathbf{G}\cdot\mathbf{r}} \phi_i(\mathbf{r}) \phi_j(\mathbf{r})$$

Here are some examples to initialize FFTDF object:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
>>> fftdf = df.FFTDF(cell)
>>> print(fftdf)
<pyscf.pbc.df.fft.FFTDF object at 0x7f599dbd6450>
>>> mf = scf.RHF(cell)
>>> print(mf.with_df)
<pyscf.pbc.df.fft.FFTDF object at 0x7f59a1a10c50>
```

As the default integral scheme of PBC calculations, FFTDF is created when initializing the PBC mean-field object and held in the attribute `with_df`.

Nuclear type integrals

PBC nuclear-electron interaction and pseudo-potential (PP) integrals can be computed with the FFTDF methods `FFTDF.get_nuc()` and `FFTDF.get_pp()`. `FFTDF.get_nuc()` function only evaluates the integral of the point charge. If PP was specified in the cell object, `FFTDF.get_nuc()` produces the integrals of the point nuclei with the effective charges. If PP was not defined in the cell object, `FFTDF.get_pp()` and `FFTDF.get_nuc()` produce the same integrals. Depending on the input k-point(s), the two functions can produce the nuclear-type integrals for a single k-point or a list of nuclear-type integrals for the k-points. By default, they compute the nuclear-type integrals of Gamma point:

```
>>> vnuc = fftdf.get_pp()
>>> print(vnuc.shape)
(2, 2)
>>> kpts = cell.make_kpts([2,2,2])
>>> vnuc = fftdf.get_pp(kpts)
>>> print(vnuc.shape)
(8, 2, 2)
>>> vnuc = fftdf.get_pp(kpts)
>>> print(vnuc.shape)
(2, 2)
```

Hartree-Fock Coulomb and exchange

`FFTDF` class provides a method `FFTDF.get_jk()` to compute Hartree-Fock Coulomb matrix (J) and exchange matrix (K). This method can take one density matrix or a list of density matrices as input and return the J and K matrices for each density matrix:

```
>>> dm = numpy.random.random((2,2))
>>> j, k = fftdf.get_jk(dm)
>>> print(j.shape)
(2, 2)
>>> dm = numpy.random.random((3,2,2))
>>> j, k = fftdf.get_jk(dm)
>>> print(j.shape)
(3, 2, 2)
```

When k-points are specified, the input density matrices should have the correct shape that matches the number of k-points:

```
>>> kpts = cell.make_kpts([1,1,3])
>>> dm = numpy.random.random((3,2,2))
```

```
>>> j, k = fftdf.get_jk(dm, kpts=kpts)
>>> print(j.shape)
(3, 2, 2)
>>> dm = numpy.random.random((5,3,2,2))
>>> j, k = fftdf.get_jk(dm, kpts=kpts)
>>> print(j.shape)
(5, 3, 2, 2)
```

4-index ERI tensor and integral transformation

4-index electron repulsion integrals can be computed with `FFTDF.get_eri()` and `FFTDF.ao2mo()` methods. Given 4 k-points(s) (corresponding to the 4 AO indices), `FFTDF.get_eri()` method produce the regular 4-index ERIs $(ij|kl)$ in AO basis. The 4 k-points should follow the law of momentum conservation

$$(\mathbf{k}_j - \mathbf{k}_i + \mathbf{k}_l - \mathbf{k}_k) \cdot a = 2n\pi.$$

By default, four Γ-points are assigned to the four AO indices. As the format of molecular ERI tensor, the PBC ERI tensor is reshaped to a 2D array:

```
>>> eri = fftdf.get_eri()
>>> print(eri.shape)
(4, 4)
>>> eri = fftdf.get_eri([kpts[0],kpts[0],kpts[1],kpts[1]])
>>> print(eri.shape)
(4, 4)
```

`FFTDF.ao2mo()` function applies integral transformation for the given four sets of orbital coefficients, four input k-points. The four k-points need to follow the momentum conservation law. Similar to `FFTDF.get_eri()`, the returned integral tensor is shaped to a 2D array:

```
>>> orbs = numpy.random.random((4,2,2))
>>> eri_mo = fftdf.get_eri(orbs, [kpts[0],kpts[0],kpts[1],kpts[1]])
>>> print(eri_mo.shape)
(4, 4)
```

Kinetic energy cutoff

The accuracy of FFTDF integrals are affected by the kinetic energy cutoff. The default kinetic energy cutoff is a conservative estimation based on the basis set and the lattice parameter. You can adjust the attribute `FFTDF.gs` (the numbers of grid points in each positive direction) to change the kinetic energy cutoff. If any values in `FFTDF.gs` is too small to reach the required accuracy `cell.precision`, `FFTDF` may output a warning message, eg:

```
WARN: ke_cutoff/gs (12.437 / [3, 4, 4]) is not enough for FFTDF to get integral␣
↪accuracy 1e-08.
Coulomb integral error is ~ 2.6 Eh.
Recomended ke_cutoff/gs are 538.542 / [20 20 20].
```

In this warning message, `Coulomb integral error` is a rough estimation for the largest error of the matrix elements of the two-electron Coulomb integrals. The overall computational error may be varied by 1 - 2 orders of magnitude.

AFTDF — AFT-based density fitting

AFTDF mans that the Fourier transform of the orbital pair is computed analytically

$$\rho_{ij}(\mathbf{G}) = \int e^{-\mathbf{G}\cdot\mathbf{r}}\phi_i(\mathbf{r})\phi_j(\mathbf{r})d^3\mathbf{r}$$

To enable AFTDF in the calculation, `AFTDF` object can be initialized and assigned to `with_df` object of mean-field object:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
>>> aft = df.AFTDF(cell)
>>> print(aft)
<pyscf.pbc.df.aft.AFTDF object at 0x7ff8b1893d90>
>>> mf = scf.RHF(cell)
>>> mf.with_df = aft
```

Generally, AFTDF is slower than FFTDF method.

`AFTDF` class offers the same methods as the `FFTDF` class. Nuclear and PP integrals, Hartree-Fock J and K matrices, electron repulsion integrals and integral transformation can be computed with functions `AFTDF.get_nuc()`, `AFTDF.get_pp()`, `AFTDF.get_jk()`, `AFTDF.get_eri()` and `AFTDF.ao2mo()` using the same calling APIs as the analogy functions in *FFTDF — FFT-based density fitting*.

Kinetic energy cutoff

`AFTDF` also makes estimation on the kinetic energy cutoff. When the any values of `AFTDF.gs` are too small for required accuracy `cell.precision`, this class also outputs the `Coulomb integral error` warning message as the `FFTDF` class.

GDF — Gaussian density fitting

GDF is an analogy of the conventional density fitting method with periodic boundary condition. The auxiliary fitting basis in PBC GDF is periodic Gaussian function (To ensure the long range Coulomb integrals converging in the real space lattice summation, the multipoles are removed from the auxiliary basis). `GDF` object can be initialized and enabled in the SCF calculation in two ways:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
>>> gdf = df.GDF(cell)
>>> mf = scf.RHF(cell)
>>> mf.with_df = gdf
>>> mf.run()
>>> # Using SCF.density_fit method
>>> mf = scf.RHF(cell).density_fit().run()
>>> print(mf.with_df)
<pyscf.pbc.df.df.GDF object at 0x7fec7722aa10>
```

Similar to the molecular code, `SCF.density_fit()` method returns a mean-field object with `GDF` as the integral engine.

In the GDF method, the DF-integral tensor is precomputed and stored on disk. GDF method supports both the Γ-point ERIs and the ERIs of different k-points. GDF.kpts should be specified before initializing GDF object. GDF class provides the same APIs as the FFTDF class to compute nuclear integrals and electron Coulomb repulsion integrals:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
>>> gdf = df.GDF(cell)
>>> gdf.kpts = cell.make_kpts([2,2,2])
>>> gdf.get_eri([kpts[0],kpts[0],kpts[1],kpts[1]])
```

In the mean-field calculation, assigning kpts attribute to mean-field object updates the kpts attribute of the underlying DF method:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
>>> mf = scf.KRHF(cell).density_fit()
>>> kpts = cell.make_kpts([2,2,2])
>>> mf.kpts = kpts
>>> mf.with_df.get_eri([kpts[0],kpts[0],kpts[1],kpts[1]])
```

Once the GDF integral tensor was initialized, the GDF can be only used with certain k-points calculations. An incorrect kpts argument can lead to a runtime error:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
>>> gdf = df.GDF(cell, kpts=cell.make_kpts([2,2,2]))
>>> kpt = np.random.random(3)
>>> gdf.get_eri([kpt,kpt,kpt,kpt])
RuntimeError: j3c for kpts [[ 0.53135523  0.06389596  0.19441766]
 [ 0.53135523  0.06389596  0.19441766]] is not initialized.
You need to update the attribute .kpts then call .build() to initialize j3c.
```

The GDF initialization is very expensive. To reduce the initialization cost in a series of calculations, it would be useful to cache the GDF integral tensor in a file then load them into the calculation when needed. The GDF integral tensor can be saved and loaded the same way as we did for the molecular DF method (see *Saving/Loading DF integral tensor*):

```
import numpy as np
from pyscf.pbc import gto, df, scf
cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
gdf = df.GDF(cell, kpts=cell.make_kpts([2,2,2]))
gdf._cderi_to_save = 'df_ints.h5'  # To save the GDF integrals
gdf.build()

mf = scf.KRHF(cell, kpts=cell.make_kpts([2,2,2])).density_fit()
mf.with_df._cderi = 'df_ints.h5'   # To load the GDF integrals
mf.run()
```

Auxiliary Gaussian basis

GDF method requires a set of Gaussian functions as the density fitting auxiliary basis. See also *DF auxiliary basis* and *Even-tempered auxiliary Gaussian basis* for the choices of DF auxiliary basis in PySCF GDF code. There are not many optimized auxiliary basis sets available for PBC AO basis. You can use the even-tempered Gaussian functions as the auxiliary basis in the PBC GDF method:

```
import numpy as np
from pyscf.pbc import gto, df, scf
cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
gdf = df.GDF(cell, kpts=cell.make_kpts([2,2,2]))
gdf.auxbasis = df.aug_etb(cell, beta=2.0)
gdf.build()
```

Kinetic energy cutoff

GDF method does not require the specification of kinetic energy cutoff. `cell.ke_cutoff` and `cell.gs` are ignored in the `GDF` class. Internally, a small set of planewaves is used in the GDF method to accelerate the convergence of GDF integrals in the real space lattice summation. The estimated energy cutoff is generated in the `GDF` class and stored in the attribute `GDF.gs`. It is not recommended to change this parameter.

MDF — mixed density fitting

MDF method combines the AFTDF and GDF in the same framework. The MDF auxiliary basis is Gaussian and plane-wave mixed basis. `MDF` object can be created in two ways:

```
>>> import numpy as np
>>> from pyscf.pbc import gto, df, scf
>>> cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g', ke_cutoff=10)
>>> mdf = df.MDF(cell)
>>> print(mdf)
<pyscf.pbc.df.mdf.MDF object at 0x7f4025120a10>
>>> mf = scf.RHF(cell).mix_density_fit().run()
>>> print(mf.with_df)
<pyscf.pbc.df.mdf.MDF object at 0x7f7963390a10>
```

The kinetic energy cutoff is specified in this example to constrain the number of planewaves. The number of planewaves can also be controlled by through attribute `MDF.gs`.

In principle, the accuracy of MDF method can be increased by adding more plane waves in the auxiliary basis. In practice, the linear dependency between plane waves and Gaussians may lead to numerical stability issue. The optimal accuracy (with reasonable computational cost) requires a reasonable size of plan wave basis with a reasonable linear dependency threshold. A threshold too large would remove many auxiliary functions while a threshold too small would cause numerical instability. .. In our preliminary test, `ke_cutoff=10` is able to produce 0.1 mEh accuracy in .. total energy. The default linear dependency threshold is 1e-10. The threshold can be adjusted through the attribute `MDF.linear_dep_threshold`.

Like the GDF method, it is also very demanding to initialize the 3-center Gaussian integrals in the MDF method. The 3-center Gaussian integral tensor can be cached in a file and loaded to `MDF` object at the runtime:

```
import numpy as np
from pyscf.pbc import gto, df, scf
cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g')
mdf = df.MDF(cell, kpts=cell.make_kpts([2,2,2]))
mdf._cderi_to_save = 'df_ints.h5'  # To save the GDF integrals
mdf.build()

mf = scf.KRHF(cell, kpts=cell.make_kpts([2,2,2])).mix_density_fit()
mf.with_df._cderi = 'df_ints.h5'   # To load the GDF integrals
mf.run()
```

All-electron calculation

All-electron calculations with FFTDF or AFTDF methods requires high energy cutoff for most elements. It is recommended to use GDF or MDF methods in the all-electron calculations. In fact, GDF and MDF can also be used in PP calculations to reduce the number of planewave basis if steep functions are existed in the AO basis.

Low-dimension system

`AFTDF` supports the systems with 0D (molecule), 1D and 2D periodic boundary conditions. When computing the integrals of low-dimension systems, an infinite vacuum is placed on the free boundary. You can set the `cell.dimension`, to enable the integral algorithms for low-dimension systems in `AFTDF` class:

```
import numpy as np
from pyscf.pbc import gto, df, scf
cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g', dimension=1)
aft = df.AFTDF(cell)
aft.get_eri()
```

`GDF` and `MDF` all support the integrals of low-dimension system. Similar to the usage of AFTDF method, you need to set `cell.dimension` for the low-dimension systems:

```
import numpy as np
from pyscf.pbc import gto, df, scf
cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g', dimension=1)
gdf = df.GDF(cell)
gdf.get_eri()
```

See more examples in `examples/pbc/31-low_dimensional_pbc.py`

Interface to molecular DF-post-HF methods

PBC DF object is compatible to the molecular DF object. The Γ-point PBC SCF object can be directly passed to molecular DF post-HF methods for an electron correlation calculations in PBC:

```
import numpy as np
from pyscf.pbc import gto, df, scf
from pyscf import cc as mol_cc
cell = gto.M(atom='He 1 1 1', a=np.eye(3)*2, basis='3-21g', dimension=1)
mf = scf.RHF(cell).density_fit()
mol_cc.RCCSD(mf).run()
```

Examples

DF relevant examples can be found in the PySCF examples directory:

```
examples/pbc/10-gamma_point_scf.py
examples/pbc/11-gamma_point_all_electron_scf.py
examples/pbc/12-gamma_point_post_hf.py
examples/pbc/20-k_points_scf.py
examples/pbc/21-k_points_all_electron_scf.py
examples/pbc/30-ao_integrals.py
examples/pbc/30-ao_value_on_grid.py
examples/pbc/30-mo_integrals.py
examples/pbc/31-low_dimensional_pbc.py
```

Program reference

FFTDF class

class pyscf.pbc.df.fft.**FFTDF**(*cell*, *kpts=array([[0., 0., 0.]])*)
Density expansion on plane waves

FFTDF helper functions

JK with discrete Fourier transformation Integral transformation with FFT

(ij|kl) = int dr1 dr2 i*(r1) j(r1) v(r12) k*(r2) l(r2) = (ij|G) v(G) (G|kl)

i*(r) j(r) = 1/N sum_G e^{iGr} (G|ij) = 1/N sum_G e^{-iGr} (ij|G)

"forward" FFT: (G|ij) = sum_r e^{-iGr} i*(r) j(r) = fft[i*(r) j(r)]

"inverse" FFT:

(ij|G) = sum_r e^{iGr} i*(r) j(r) = N * ifft[i*(r) j(r)] = conj[sum_r e^{-iGr} j*(r) i(r)]

AFTDF class

class pyscf.pbc.df.aft.**AFTDF**(*cell*, *kpts=array([[0., 0., 0.]])*)
Density expansion on plane waves

AFTDF helper functions

JK with analytic Fourier transformation Integral transformation with analytic Fourier transformation

GDF class

class pyscf.pbc.df.df.**GDF**(*cell*, *kpts=array([[0., 0., 0.]])*)
Gaussian density fitting

GDF helper functions

Density fitting with Gaussian basis Ref:

MDF class

class pyscf.pbc.df.mdf.**MDF**(*cell*, *kpts=array([[0., 0., 0.]])*)
Gaussian and planewaves mixed density fitting

MDF helper functions

Exact density fitting with Gaussian and planewaves Ref:

1.21.5 pbc.cc — PBC coupled cluster

1.21.6 pbc.tools — PBC tools

Interface to ASE

The ASE (Atomic Simulation Environment) tool set offers useful database and functions to setup crystal structure and analyze the results of crystal calculation.

Here are some examples to use the PySCF-ASE interface wrapper

```
"""
Take ASE Diamond structure, input into PySCF and run
"""

import numpy as np
import pyscf.pbc.gto as pbcgto
import pyscf.pbc.dft as pbcdft
from pyscf.pbc.tools import pyscf_ase

import ase
import ase.lattice
from ase.lattice.cubic import Diamond


ase_atom=Diamond(symbol='C', latticeconstant=3.5668)
print(ase_atom.get_volume())

cell = pbcgto.Cell()
cell.verbose = 5
cell.atom=pyscf_ase.ase_atoms_to_pyscf(ase_atom)
cell.a=ase_atom.cell
cell.basis = 'gth-szv'
cell.pseudo = 'gth-pade'
cell.build()

mf=pbcdft.RKS(cell)

mf.xc='lda,vwn'

print(mf.scf()) # [10,10,10]: -44.8811199336
```

```
"""
Take ASE structure, PySCF object,
and run through ASE calculator interface.

This allows other ASE methods to be used with PySCF;
here we try to compute an equation of state.
"""

import numpy as np
from pyscf.pbc.tools import pyscf_ase
import pyscf.pbc.gto as pbcgto
import pyscf.pbc.dft as pbcdft

import ase
import ase.lattice
```

```
from ase.lattice.cubic import Diamond
from ase.units import kJ
from ase.utils.eos import EquationOfState


ase_atom=Diamond(symbol='C', latticeconstant=3.5668)

# Set up a cell; everything except atom; the ASE calculator will
# set the atom variable
cell = pbcgto.Cell()
cell.a=ase_atom.cell
cell.basis = 'gth-szv'
cell.pseudo = 'gth-pade'
cell.verbose = 0

# Set up the kind of calculation to be done
# Additional variables for mf_class are passed through mf_dict
mf_class=pbcdft.RKS
mf_dict = { 'xc' : 'lda,vwn' }

# Once this is setup, ASE is used for everything from this point on
ase_atom.set_calculator(pyscf_ase.PySCF(molcell=cell, mf_class=mf_class, mf_dict=mf_
↪dict))

print("ASE energy", ase_atom.get_potential_energy())
print("ASE energy (should avoid re-evaluation)", ase_atom.get_potential_energy())
# Compute equation of state
ase_cell=ase_atom.cell
volumes = []
energies = []
for x in np.linspace(0.95, 1.2, 5):
    ase_atom.set_cell(ase_cell * x, scale_atoms = True)
    print "[x: %f, E: %f]" % (x, ase_atom.get_potential_energy())
    volumes.append(ase_atom.get_volume())
    energies.append(ase_atom.get_potential_energy())

eos = EquationOfState(volumes, energies)
v0, e0, B = eos.fit()
print(B / kJ * 1.0e24, 'GPa')
eos.plot('eos.png')
```

1.21.7 Mixing PBC and molecular modules

The post-HF methods, as a standalone numerical solver, do not require the knowledge of the boundary condition. The calculations of finite-size systems and extend systems are distinguished by the boundary condition of integrals (and basis). The same post-HF solver can be used for both the finite-size problem and the periodic boundary problem if they have the similar Hamiltonian structure.

In PySCF, many molecular post-HF solvers has two implementations: incore and outcore versions. They are differed by the treatments on the 2-electron integrals. The incore solver takes the `_eri` (or `with_df`, see *df — Density fitting*) from the underlying mean-field object as the two-electron interaction part of the Hamiltonian while the outcore solver generates the 2-electron integrals (with free boundary condition) on the fly. To use the molecular post-HF solvers in PBC code, we need ensure the incore version solver being called.

Generating `_eri` in mean-filed object is the straightforward way to trigger the incore post-HF solver. If the allowed memory is big enough to hold the entire 2-electron integral array, the gamma point HF solver always generates and

holds this array. A second choice is to set `incore_anyway` in `cell` which forces the program generating and holding `_eri` in mean-field object.

Note: If the problem is big, `incore_anyway` may overflow the available physical memory.

Holding the full integral array `_eri` in memory limits the problem size one can treat. Using the density fitting object `with_df` to hold the integrals can overcome this problem. This architecture has been bound to PBC and molecular mean-field modules. But the relevant post-HF density fitting solvers are still in development thus this feature is not available in PySCF 1.2 or older.

Aside from the 2-electron integrals, there are some attributes and methods required by the post-HF solver. They are `get_hcore()`, and `get_ovlp()` for 1-electron integrals, `_numint`, `grids` for the numerical integration of DFT exchange-correlation functionals. They are all overloaded in PBC mean-field object to produce the PBC integrals.

Examples

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

'''
Gamma point post-HF calculation needs only real integrals.
Methods implemented in finite-size system can be directly used here without
any modification.
'''

import numpy
from pyscf.pbc import gto, scf

cell = gto.M(
    a = numpy.eye(3)*3.5668,
    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 = '6-31g',
    verbose = 4,
)

mf = scf.RHF(cell).density_fit()
mf.with_df.gs = [5]*3
mf.kernel()

#
# Import CC, TDDFT moduel from the molecular implementations
#
from pyscf import cc, tddft
mycc = cc.CCSD(mf)
mycc.kernel()

mytd = tddft.TDHF(mf)
mytd.nstates = 5
mytd.kernel()
```

1.22 lo — Orbital localization

1.22.1 Foster-Boys, Edmiston-Ruedenberg, Pipek-Mezey localization

pyscf.lo.pipek.**atomic_pops**(*mol*, *mo_coeff*, *method='meta_lowdin'*)
kwarg method can be one of mulliken, lowdin, meta_lowdin

1.22.2 Meta-Lowdin

1.22.3 Natural atomic orbitals

Natural atomic orbitals Ref:

6. Weinhold et al., J. Chem. Phys. 83(1985), 735-746

pyscf.lo.nao.**set_atom_conf**(*element*, *description*)
Change the default atomic core and valence configuration to the one given by "description". See lo.nao.AOSHELL for the default configuration.

Args:

element [str or int] Element symbol or nuclear charge

description [str or a list of str]

"double p" : double p shell
"double d" : double d shell
"double f" : double f shell
"polarize" : add one polarized shell
"1s1d" : keep core unchanged and set 1 s 1 d shells for valence
("3s2p","1d") : 3 s, 2 p shells for core and 1 d shells for valence

1.22.4 Intrinsic Atomic Orbitals

Intrinsic Atomic Orbitals ref. JCTC, 9, 4834

pyscf.lo.iao.**iao**(*mol*, *orbocc*, *minao='minao'*)
Intrinsic Atomic Orbitals. [Ref. JCTC, 9, 4834

Args:

orbocc [2D float array] occupied orbitals

Returns: non-orthogal IAO orbitals. Orthogonalize them as C (C^T S C)^{-1/2}, eg using `orth.lowdin()`

```
>>> orbocc = mf.mo_coeff[:,mf.mo_occ>0]
>>> c = iao(mol, orcc)
>>> numpy.dot(c, orth.lowdin(reduce(numpy.dot, (c.T,s,c))))
```

1.23 Miscellaneous

1.23.1 Decoration pipe

SCF

There are three decoration function for Hartree-Fock class `density_fit()`, `sfx2c()`, `newton()` to apply density fitting, scalar relativistic correction and second order SCF. The different ordering of the three decoration operations have different effects. For example

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

import numpy
from pyscf import gto
from pyscf import scf

'''
Mixing decoration, for density fitting, scalar relativistic effects, and
second order (Newton-Raphson) SCF.

Density fitting and scalar relativistic effects can be applied together,
regardless to the order you apply the decoration.

NOTE the second order SCF (New in version 1.1) decorating operation are not
commutable with scf.density_fit operation
        [scf.density_fit, scf.sfx2c      ] == 0
        [scf.newton     , scf.sfx2c      ] == 0
        [scf.newton     , scf.density_fit] != 0
* scf.density_fit(scf.newton(scf.RHF(mol))) is the SOSCF for regular 2e
  integrals, but with density fitting integrals for the Hessian.  It's an
  approximate SOSCF optimization method;
* scf.newton(scf.density_fit(scf.RHF(mol))) is the exact second order
  optimization for the given scf object which is a density-fitted-scf method.
  The SOSCF is not an approximate scheme.
* scf.density_fit(scf.newton(scf.density_fit(scf.RHF(mol))), auxbasis='ahlrichs')
  is an approximate SOSCF scheme for the given density-fitted-scf method.
  Here we use small density fitting basis (ahlrichs cfit basis) to approximate
  the Hessian for the large-basis-density-fitted-scf scheme.
'''

mol = gto.Mole()
mol.build(
    verbose = 0,
    atom = '''8  0  0.     0
              1  0  -0.757 0.587
              1  0  0.757  0.587''',
    basis = 'ccpvdz',
)

#
# 1. spin-free X2C-HF with density fitting approximation on 2E integrals
#
mf = scf.density_fit(scf.sfx2c(scf.RHF(mol)))
mf = scf.RHF(mol).x2c().density_fit()  # Stream style
```

```
energy = mf.kernel()
print('E = %.12f, ref = -76.075408156180' % energy)

#
# 2. spin-free X2C correction for density-fitting HF.  Since X2C correction is
# commutable with density fitting operation, it is fully equivalent to case 1.
#
mf = scf.sfx2c(scf.density_fit(scf.RHF(mol)))
mf = scf.RHF(mol).density_fit().x2c()  # Stream style
energy = mf.kernel()
print('E = %.12f, ref = -76.075408156180' % energy)

#
# 3. Newton method for non-relativistic HF
#
mf = scf.newton(scf.RHF(mol))
mf = scf.RHF(mol).newton()  # Stream style
energy = mf.kernel()
print('E = %.12f, ref = -76.026765673120' % energy)

#
# 4. Newton method for non-relativistic HF with density fitting for orbital
# hessian of newton solver.  Note the answer is equal to case 3, but the
# solver "mf" is different.
#
mf = scf.density_fit(scf.newton(scf.RHF(mol)))
mf = scf.RHF(mol).newton().density_fit()
energy = mf.kernel()
print('E = %.12f, ref = -76.026765673120' % energy)

#
# 5. Newton method to solve the density-fitting approximated HF object.  There
# is no approximation for newton method (orbital hessian).  Note the density
# fitting is applied on HF object only.  It does not affect the Newton solver.
#
mf = scf.newton(scf.density_fit(scf.RHF(mol)))
mf = scf.RHF(mol).density_fit().newton()
energy = mf.kernel()
print('E = %.12f, ref = -76.026744737357' % energy)

#
# 6. Newton method for density-fitting HF, and the hessian of Newton solver is
# also approximated with density fitting.  Note the anwser is equivalent to
# case 5, but the solver "mf" is different.  Here the fitting basis for HF and
# Newton solver are different.  HF is approximated with the default density
# fitting basis (Weigend cfit basis).  Newton solver is approximated with
# Ahlrichs cfit basis.
#
mf = scf.density_fit(scf.newton(scf.density_fit(scf.RHF(mol))), 'ahlrichs')
mf = scf.RHF(mol).density_fit().newton().density_fit(auxbasis='ahlrichs')
energy = mf.kernel()
print('E = %.12f, ref = -76.026744737357' % energy)
```

FCI

Direct FCI solver cannot guarantee the CI wave function to be the spin eigenfunction. Decoration function `fci.addons.fix_spin_()` can fix this issue.

CASSCF

`mcscf.density_fit()`, and `scf.sfx2c()` can be used to decorate CASSCF/CASCI class. Like the ordering problem in SCF decoration operation, the density fitting for CASSCF solver only affect the CASSCF optimization procedure. It does not change the 2e integrals for CASSCF Hamiltonian. For example

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

from pyscf import gto, scf, mcscf

'''
Density fitting for orbital optimzation.

Note mcscf.density_fit function follows the same convention of decoration
ordering which is applied in the SCF decoration.  See pyscf/mcscf/df.py for
more details and pyscf/example/scf/23-decorate_scf.py as an exmple.
'''

mol = gto.Mole()
mol.build(
    atom = [
    ["C", (-0.65830719,  0.61123287, -0.00800148)],
    ["C", ( 0.73685281,  0.61123287, -0.00800148)],
    ["C", ( 1.43439081,  1.81898387, -0.00800148)],
    ["C", ( 0.73673681,  3.02749287, -0.00920048)],
    ["C", (-0.65808819,  3.02741487, -0.00967948)],
    ["C", (-1.35568919,  1.81920887, -0.00868348)],
    ["H", (-1.20806619, -0.34108413, -0.00755148)],
    ["H", ( 1.28636081, -0.34128013, -0.00668648)],
    ["H", ( 2.53407081,  1.81906387, -0.00736748)],
    ["H", ( 1.28693681,  3.97963587, -0.00925948)],
    ["H", (-1.20821019,  3.97969587, -0.01063248)],
    ["H", (-2.45529319,  1.81939187, -0.00886348)],],
    basis = 'ccpvtz'
)

mf = scf.RHF(mol)
mf.conv_tol = 1e-8
e = mf.kernel()

#
# DFCASSCF uses density-fitting 2e integrals overall, regardless the
# underlying mean-filed object
#
mc = mcscf.DFCASSCF(mf, 6, 6)
mo = mc.sort_mo([17,20,21,22,23,30])
mc.kernel(mo)
print('E(CAS) = %.12f, ref = -230.845892901370' % mc.e_tot)
```

```
#
# Assign DF basis
#
mc = mcscf.DFCASSCF(mf, 6, 6, auxbasis='ccpvtzfit')
mo = mc.sort_mo([17,20,21,22,23,30])
mc.kernel(mo)
print('E(CAS) = %.12f, ref = -230.845892901370' % mc.e_tot)
```

1.23.2 Customizing Hamiltonian

PySCF supports user-defined Hamiltonian for many modules. To customize Hamiltonian for Hartree-Fock, CASSCF, MP2, CCSD, etc, one need to replace the methods `get_hcore()`, `get_ovlp()` and attribute `_eri` of SCF class for new Hamiltonian. E.g. the user-defined Hamiltonian for Hartree-Fock

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

import numpy
from pyscf import gto, scf, ao2mo

'''
Customizing Hamiltonian for SCF module.

Three steps to define Hamiltonian for SCF:
1. Specify the number of electrons. (Note mole object must be "built" before doing
 ↪this step)
2. Overwrite three attributes of scf object
    .get_hcore
    .get_ovlp
    ._eri
3. Specify initial guess (to overwrite the default atomic density initial guess)

Note you will see warning message on the screen:

        overwrite keys get_ovlp get_hcore of <class 'pyscf.scf.hf.RHF'>
'''

mol = gto.M()
n = 10
mol.nelectron = n

mf = scf.RHF(mol)
h1 = numpy.zeros((n,n))
for i in range(n-1):
    h1[i,i+1] = h1[i+1,i] = -1.0
h1[n-1,0] = h1[0,n-1] = -1.0  # PBC
eri = numpy.zeros((n,n,n,n))
for i in range(n):
    eri[i,i,i,i] = 4.0

mf.get_hcore = lambda *args: h1
mf.get_ovlp = lambda *args: numpy.eye(n)
# ao2mo.restore(8, eri, n) to get 8-fold permutation symmetry of the integrals
```

```
# ._eri only supports the two-electron integrals in 4-fold or 8-fold symmetry.
mf._eri = ao2mo.restore(8, eri, n)

mf.kernel()
```

and the user-defined Hamiltonian for CASSCF

```
#!/usr/bin/env python
#
# Author: Qiming Sun <osirpt.sun@gmail.com>
#

import numpy
from pyscf import gto, scf, ao2mo, mcscf

'''
User-defined Hamiltonian for CASSCF module.

Defining Hamiltonian once for SCF object, the derivate post-HF method get the
Hamiltonian automatically.
'''

mol = gto.M()
mol.nelectron = 6

#
# 1D anti-PBC Hubbard model at half filling
#
n = 12

h1 = numpy.zeros((n,n))
for i in range(n-1):
    h1[i,i+1] = h1[i+1,i] = -1.0
h1[n-1,0] = h1[0,n-1] = -1.0
eri = numpy.zeros((n,n,n,n))
for i in range(n):
    eri[i,i,i,i] = 2.0

mf = scf.RHF(mol)
mf.get_hcore = lambda *args: h1
mf.get_ovlp = lambda *args: numpy.eye(n)
mf._eri = ao2mo.restore(8, eri, n)
mf.init_guess = '1e'
mf.kernel()

mycas = mcscf.CASSCF(mf, 4, 4)
mycas.kernel()
```

1.24 qmmm — QM/MM interface

QM part interface

pyscf.qmmm.itrf.**mm_charge**(*scf_method*, *coords*, *charges*, *unit=None*)

Modify the QM method using the (non-relativistic) potential generated by MM charges.

Args: scf_method : a HF or DFT object

coords [2D array, shape (N,3)] MM particle coordinates

charges [1D array] MM particle charges

Kwargs:

unit [str] Bohr, AU, Ang (case insensitive). Default is the same to mol.unit

Returns: Same method object as the input scf_method with modified 1e Hamiltonian

Note: 1. if MM charge and X2C correction are used together, function mm_charge needs to be applied after X2C decoration (scf.sfx2c function), eg mf = mm_charge(scf.sfx2c(scf.RHF(mol)), [(0.5,0.6,0.8)], [-0.5]). 2. Once mm_charge function is applied on the SCF object, it affects all the post-HF calculations eg MP2, CCSD, MCSCF etc

Examples:

```
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = mm_charge(dft.RKS(mol), [(0.5,0.6,0.8)], [-0.3])
>>> mf.kernel()
-101.940495711284
```

pyscf.qmmm.itrf.**mm_charge_grad**(*scf_grad*, *coords*, *charges*, *unit=None*)

Apply the MM charges in the QM gradients' method. It affects both the electronic and nuclear parts of the QM fragment.

Args:

scf_grad [a HF or DFT gradient object (grad.HF or grad.RKS etc)] Once mm_charge_grad function is applied on the SCF object, it affects all post-HF calculations eg MP2, CCSD, MCSCF etc

coords [2D array, shape (N,3)] MM particle coordinates

charges [1D array] MM particle charges

Kwargs:

unit [str] Bohr, AU, Ang (case insensitive). Default is the same to mol.unit

Returns: Same gradeints method object as the input scf_grad method

Examples:

```
>>> from pyscf import gto, scf, grad
>>> mol = gto.M(atom='H 0 0 0; F 0 0 1', basis='ccpvdz', verbose=0)
>>> mf = mm_charge(scf.RHF(mol), [(0.5,0.6,0.8)], [-0.3])
>>> mf.kernel()
-101.940495711284
>>> hfg = mm_charge_grad(grad.hf.RHF(mf), coords, charges)
>>> hfg.kernel()
[[-0.25912357 -0.29235976 -0.38245077]
 [-1.70497052 -1.89423883  1.2794798 ]]
```

1.25 mrpt — Multi-reference perturbation theory

1.25.1 N-electron valance perturbation theory (NEVPT2)

class pyscf.mrpt.nevpt2.**NEVPT**(*mc*, *root=0*)

Strongly contracted NEVPT2

Attributes:

root [int] To control which state to compute if multiple roots or state-average wfn were calculated in CASCI/CASSCF

compressed_mps [bool] compressed MPS perturber method for DMRG-SC-NEVPT2

Examples:

```
>>> mf = gto.M('N 0 0 0; N 0 0 1.4', basis='6-31g').apply(scf.RHF).run()
>>> mc = mcscf.CASSCF(mf, 4, 4).run()
>>> NEVPT(mc).kernel()
-0.14058324991532101
```

compress_approx(*maxM=500*, *compress_schedule=None*, *tol=1e-07*, *stored_integral=False*)
SC-NEVPT2 with compressed perturber

Kwargs :

maxM [int] DMRG bond dimension

Examples:

```
>>> mf = gto.M('N 0 0 0; N 0 0 1.4', basis='6-31g').apply(scf.RHF).run()
>>> mc = dmrgscf.DMRGSCF(mf, 4, 4).run()
>>> NEVPT(mc, root=0).compress_approx(maxM=100).kernel()
-0.14058324991532101
```

load_ci(*root=None*)
Hack me to load CI wfn from disk

1.26 Benchmark

Platform	
CPU	4 Intel E5-2670 @ 2.6 GB
Memory	64 GB DDR3
OS	Custom Redhat 6.6
BLAS	MKL 11.0
Compiler	Intel 13.0

Benzene, on 16 CPU cores

Basis	6-31G**	cc-pVTZ	ANO-Roos-TZ
HF	0.55 s	5.76 s	389.1 s
density fit HF	3.56 s	7.61 s	13.8 s
B3LYP	3.84 s	11.44 s	360.2 s
MP2	0.21 s	4.66 s	115.9 s
CASSCF(6,6)	2.88 s	34.73 s	639.7 s
CCSD	18.24 s	477.0 s	6721 s

C60, on 16 CPU cores

Basis	6-31G**	cc-pVTZ
HF	1291 s	189 m
SOSCF (newton)		77 m
density fit HF	316.7 s	43.3 m

Fe(II)-porphyrin (FeC20H12N4), on 16 CPU cores

Basis	cc-pVDZ	cc-pVTZ	cc-pVQZ
SOSCF (newton)	193.4 s	20.1 m	127.1 m
CASSCF(10,10)	1808 s	241 m	
CASSCF(11,8)	763.8 s	150.3 m	1280 m

1.27 Code standard

- Code at least should work under python-2.7, gcc-4.8.
- 90/10 functional/OOP, unless performance critical, functions are pure.
- 90/10 Python/C, only computational hot spots were written in C.
- To extend python function with C/Fortran:
  - Following C89 (gnu89) standard for C code. (complex? variable length array?) http://flash-gordon.me.uk/ansi.c.txt
  - Following Fortran 95 standard for Fortran code. http://j3-fortran.org/doc/standing/archive/007/97-007r2/pdf/97-007r2.pdf
  - Do **not** use other program languages (to keep the package light-weight).
- Conservative on advanced language feature.
- Minimal dependence principle
  - Minimal requirements on 3rd party program or libraries.
  - Loose-coupling between modules so that the failure of one module can have minimal effects on the other modules.
- Not enforced but recommended - Compatibile with Python 2.6, 2.7, 3.2, 3.3, 3.4; - Following C89 (gnu89) standard for C code; - Using ctypes to bridge C/python functions

1.27.1 Name convention

- The prefix or suffix underscore in the function names have special meanings
  - functions with prefix-underscore like `_fn` are private functions. They are typically not documented, and not recommended to use.
  - functions with suffix-underscore like `fn_` means that they have side effects. The side effects include the change of the input argument, the runtime modification of the class definitions (attributes or members), or module definitions (global variables or functions) etc.
  - regular (pure) functions do not have underscore as the prefix or suffix.

1.27.2 API convention

- `gto.Mole` holds all global parameters, like the log level, the max memory usage etc. They are used as the default value for all other classes.
- Method class.
  - Most QC method classes (like HF, CASSCF, FCI, ...) directly take three attributes `verbose`, `stdout` and `max_memory` from `gto.Mole`. Overwriting them only affects the behavior of the local instance for that method class. In the following example, `mf.verbose` screens out the noises produced by `RHF` method, and the output of `MP2` is written in the log file `example.log`:

```
>>> from pyscf import gto, scf, mp
>>> mol = gto.M(atom='H 0 0 0; H 0 0 1', verbose=5)
>>> mf = scf.RHF(mol)
>>> mf.verbose = 0
>>> mf.kernel()
>>> mp2 = mp.MP2(mf)
>>> mp2.stdout = open('example.log', 'w')
>>> mp2.kernel()
```

  - Method class are only to hold the options or enviroments (like convergence threshold, max iterations, ...) to control the behavior/convergence of the method. The intermediate status are **not** supposed to be saved in the method class (during the computation). However, the final results or solutions are kept in the method object for convenience. Once the results are stored in the particular method class, they are assumed to be read only, since many class member functions take them as the default arguments if the caller didn't provide enough parameters.
  - In __init__ function, initialize/define the problem size. The problem size parameters (like num orbitals etc) can be considered as enviroments. They are not supposed to be changed by other functions.
  - Kernel functions Although the method classes have various entrance/main funtion, many of them provide an entrance function called `kernel`. You can simply call the `kernel` function and it will guide the program flow to the right main function.
  - Default value of the class member functions' arguments. Many member functions can take the results of their class as the default arguments.

- Function arguments
  - First argument is handler. The handler is one of `gto.Mole` object, a mean-field object, or a post-Hartree-Fock object.

1.28 Version history

1.4	2017-10-05
1.4 beta	2017-08-22 (feature freeze)
1.4 alpha	2017-06-24
1.3.5_	2017-08-12
1.3.4	2017-08-09
1.3.3	2017-07-05
1.3.2	2017-06-05
1.3.1	2017-05-14
1.3	2017-04-25
1.3 beta	2017-02-15 (feature freeze)
1.3 alpha 2	2017-01-04
1.3 alpha 1	2016-12-04
1.2.3	2017-04-24
1.2.2	2017-02-15
1.2.1	2017-01-26
1.2	2016-11-07
1.2 beta	2016-09-13 (feature freeze)
1.2 alpha	2016-08-05
1.1	2016-06-04
1.1 beta	2016-04-11 (feature freeze)
1.1 alpha 2	2016-03-08
1.1 alpha 1	2016-02-08
1.0	2015-10-07
1.0 rc1	2015-09-07
1.0 beta 1	2015-08-02 (feature freeze)
1.0 alpha 2	2015-07-03
1.0 alpha 1	2015-04-07

You can also download the PDF version of this manual.

c

d

p

S

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

Z